EMAIL_INBOUND Azure AI Search Integration - Design Document

📋 Overview

Complete Azure AI Search implementation for EMAIL_INBOUND with:

- Vector Embeddings for semantic search
- Attachment text extraction (TXT, DOCX, PPTX, PDF)
- Hybrid Search (Keyword + Semantic + Vector)
- Multi-Tenancy Security (BU_GUID + Project_Id mandatory filters)
- Direct indexing from Mailgun webhook (no SQL intermediary)

📐 Architecture

Flow:

```
Mailgun Webhook → Azure Function (Linux, C#, Flex Consumption)
  ↓
Parse multipart/form-data
  ↓
Extract text from attachments (parallel processing)
  ├─> TXT → Direct read
  ├─> DOCX → OpenXML
  ├─> PPTX → OpenXML
  └─> PDF → PdfPig (text) OR Azure Document Intelligence (OCR)
  ↓
Combine: Subject + Body + Attachment texts
  ↓
Generate OpenAI embedding (text-embedding-ada-002)
  ↓
Index in Azure AI Search
  ↓
Return 200 OK to Mailgun
```

Processing Time:

- Without OCR: < 5 seconds
- With OCR: < 15 seconds (90% under 30 seconds)

📊 Index Schema

Core Principles:

1. **Security-First**: BU_GUID + Project_Id as MANDATORY filters
2. **Hybrid Search**: Keyword + Semantic + Vector combined
3. **German Language**: Optimized for German construction texts
4. **Attachment Content**: Searchable including extracted text from PDFs
5. **NO Chunking**: Emails are small enough for single document

---

Field Structure:

1. Identification & Security (MANDATORY Filters)

```
json

{
  "id": "email-guid",                  // Primary Key
  "email_guid": "xxx-xxx-xxx",         // Reference GUID
  "bu_guid": "A1B2C3D4-...",           // ✅ MANDATORY FILTER
  "project_id": 14262,                 // ✅ MANDATORY FILTER
  "project_name": "Construction Hall 5"  // Searchable
}
```

Critical:

- `bu_guid` + `project_id` must ALWAYS be filtered in queries
- No search allowed without these filters (Row-Level Security)

---

2. Object Context (Optional Filters)

```
json
```

```
{
  "keypoint_id": 123,      // Optional: Email linked to Keypoint
  "workitem_id": 456,      // Optional: Email linked to WorkItem
  "lesson_id": null,       // Optional: Email linked to Lesson
  "appointment_id": null,  // Optional: Email linked to Appointment
  "risk_id": null          // Optional: Email linked to Risk
}
```

Use Case: "Show me all emails for Keypoint 123"

---

3. Email Metadata (Searchable + Filterable)

```
json

{
  "sender_email": "ibrahim.azim@semyou.com",  // Searchable + Filterable
  "sender_name": "Ibrahim Azim",              // Searchable (German analyzer)
  "recipient_email": "b1-p14262@inbox.pcpx.ai", // Filterable
  "subject": "Construction Plan Hall 5 Update", // Searchable (German)
  "message_id": "AS8PR09MB5223CB368...",      // Filterable (duplicates)
  "created_at": "2026-01-22T10:30:00Z"       // Filterable + Sortable
}
```

---

4. Content Fields (Searchable, NO Filters)

A. Body Text

```
json

{
  "body_text": "Extracted text from body-plain or body-html"
}
```

Processing Logic:

```
IF body-plain exists AND length > 100:
  body_text = body-plain  ✅ Prefer plain text
ELSE IF body-html exists:
  body_text = StripHtml(body-html)  ✅ Convert HTML to text
ELSE:
  body_text = ""
```

Why not both?

- Azure Search Limit: 32 KB per field
- HTML is redundant when Plain exists
- Search works better on Plain (no HTML tags)

B. Combined Content

```
json

{
  "content_combined": "Subject + Body + Attachment Texts"
}
```

Structure:

```
Subject: Construction Plan Hall 5 Update

Body:
Please review the attached updated plans for Hall 5.

Attachments:
[Construction_Plan_Hall5_v2.pdf]
Foundation: Depth 2.5m, Material C30/37...

[Calculation.xlsx]
Item | Quantity | Price
Concrete | 50m³ | €8500
```

Advantages:

- **Single vector embedding** for entire email content
- **Better semantic search** (context from Subject + Body + Attachments)

- **No chunking needed** (see below)

---

5. Vector Embeddings

```json
{
  "content_vector": [0.0123, -0.0456, 0.0789, ..., 0.0234]  // 1536 dimensions
}
```

Generation:

```csharp
// Combine all content
var combinedText = $"{subject}\n\n{bodyText}";

if (attachmentTexts.Any())
{
    combinedText += "\n\nAttachments:\n" + string.Join("\n\n", attachmentTexts);
}

// Truncate if too long (OpenAI limit: ~8000 tokens = ~30,000 chars)
if (combinedText.Length > 30000)
{
    combinedText = combinedText.Substring(0, 30000);
}

// Generate embedding
var embedding = await OpenAIClient.GetEmbeddingsAsync(combinedText);
```

---

6. Attachments

```json
```

```
{
  "has_attachments": true,
  "attachment_count": 3,
  "attachment_names": [
    "Construction_Plan_Hall5_v2.pdf",
    "Calculation.xlsx",
    "Photo_Foundation.jpg"
  ],
  "attachment_types": [
    "application/pdf",
    "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
    "image/jpeg"
  ],
  "attachment_texts": [
    "Extracted text from Construction_Plan_Hall5_v2.pdf: Foundation Depth 2.5m...",
    "Extracted text from Calculation.xlsx: Concrete 50m³ €8500...",
    ""  // No text from image (unless OCR)
  ]
}
```

Processing:

- `attachment_names`: **Searchable** → "Find email with construction plan.pdf"
- `attachment_types`: **Filterable** → "Only emails with PDFs"
- `attachment_texts`: **Searchable** → "Search in attachment content"

🔄 Processing Pipeline

Phase 1: Mailgun Webhook Arrives

```
Mailgun POST → Azure Function (MailgunInbound)
  ├─> Parse multipart/form-data
  ├─> Extract fields:
  │   ├─> subject
  │   ├─> sender
  │   ├─> from (sender name)
  │   ├─> body-plain
  │   ├─> body-html
  │   ├─> attachment-count
  │   └─> attachment-1, attachment-2, ...
```

```
└─> Parse recipient: b1-p14262-k123@inbox.pcpx.ai
    ├─> Business_Id: 1
    ├─> Project_Id: 14262
    └─> Keypoint_Id: 123 (optional)
```

Phase 2: Process Attachments (PARALLEL)

```csharp
var attachmentTasks = new List<Task<AttachmentInfo>>();

for (int i = 1; i <= attachmentCount; i++)
{
    var attachmentFile = mailgunData[$"attachment-{i}"];

    attachmentTasks.Add(Task.Run(async () =>
    {
        // Extract text from attachment
        var extractedText = await ExtractTextFromAttachmentAsync(attachmentFile);

        return new AttachmentInfo
        {
            FileName = attachmentFile.FileName,
            ContentType = attachmentFile.ContentType,
            ExtractedText = extractedText
        };
    }));
}

var attachments = await Task.WhenAll(attachmentTasks);
```

Text Extraction:

```csharp
```

```
private async Task<string> ExtractTextFromAttachmentAsync(IFormFile file)
{
    var extension = Path.GetExtension(file.FileName).ToLower();

    switch (extension)
    {
        case ".txt":
            return await ReadTextFileAsync(file);

        case ".docx":
            return ExtractFromDocx(file);

        case ".pptx":
            return ExtractFromPptx(file);

        case ".pdf":
            // Try text extraction first (FREE)
            var text = ExtractTextFromPdf(file);

            if (text.Length > 200)
            {
                return text; // ✅ Text-PDF (90% of cases)
            }
            else
            {
                // Fallback: OCR with Azure Document Intelligence
                return await OcrPdfAsync(file); // ✅ Image-PDF (10% of cases)
            }

        default:
            return ""; // No text extraction for images, etc.
    }
}
```

Phase 3: Index in Azure Search

csharp

```
// Prepare body text (prefer plain, fallback to HTML)
var bodyText = !string.IsNullOrEmpty(mailgunData.BodyPlain)
    ? mailgunData.BodyPlain
    : StripHtml(mailgunData.BodyHtml ?? "");

// Combine all content for embedding
var combinedContent = $"{mailgunData.Subject}\n\n{bodyText}";

if (attachments.Any(a => !string.IsNullOrEmpty(a.ExtractedText)))
{
    combinedContent += "\n\nAttachments:\n" +
        string.Join("\n\n", attachments
            .Where(a => !string.IsNullOrEmpty(a.ExtractedText))
            .Select(a => $"[{a.FileName}]\n{a.ExtractedText}"));
}

// Truncate if too long
if (combinedContent.Length > 30000)
{
    combinedContent = combinedContent.Substring(0, 30000);
}

// Generate embedding
var embedding = await openAI.GenerateEmbeddingAsync(combinedContent);

// Get Project Name (from existing Azure Search or API)
var projectName = await GetProjectNameAsync(projectId);

// Create search document
var searchDoc = new EmailSearchDocument
{
    Id = Guid.NewGuid().ToString(),
    EmailGuid = Guid.NewGuid().ToString(),

    // MANDATORY Filters
    BuGuid = buGuid.ToString(),
    ProjectId = projectId,
    ProjectName = projectName,

    // Optional Filters
    KeypointId = keypointId,
    WorkItemId = workItemId,
    // ...
```

```
    // Searchable Fields
    SenderEmail = mailgunData.Sender,
    SenderName = ExtractName(mailgunData.From),
    RecipientEmail = mailgunData.Recipient,
    Subject = mailgunData.Subject,
    BodyText = bodyText,
    ContentCombined = combinedContent,

    // Vector
    ContentVector = embedding,

    // Attachments
    HasAttachments = attachments.Any(),
    AttachmentCount = attachments.Count,
    AttachmentNames = attachments.Select(a => a.FileName).ToList(),
    AttachmentTypes = attachments.Select(a => a.ContentType).Distinct().ToList(),
    AttachmentTexts = attachments
        .Select(a => a.ExtractedText ?? "")
        .Where(t => !string.IsNullOrEmpty(t))
        .ToList(),

    // Metadata
    MessageId = mailgunData.MessageId,
    CreatedAt = DateTimeOffset.UtcNow
};

// Index document
await searchClient.IndexDocumentsAsync(
    IndexDocumentsBatch.Upload(new[] { searchDoc })
);

// Return success to Mailgun
return new HttpResponseData(HttpStatusCode.OK);
```

❓ Chunking - YES or NO?

NO - No Chunking Needed! ✅

Reasoning:

1. Email Size is Small

Typical Email:
- Subject: 50-100 characters
- Body: 500-2000 characters
- Attachments: 3 × 1000 characters (extracted)

Total: ~3500-6500 characters

OpenAI Embedding Limit: 8191 tokens ≈ 30,000 characters
Azure Search Field Limit: 32 KB ≈ 32,000 characters

→ Fits easily! ✅

2. Email is a Logical Unit

- Users search for **entire email**, not paragraphs
- Context is lost with chunking
- Subject + Body + Attachments belong together

3. Vector Embeddings Work Better Without Chunking

One embedding for "Construction Plan Hall 5 + Foundation 2.5m + Concrete C30/37"
→ Finds semantically similar emails ✅

Three separate embeddings:
1. "Construction Plan Hall 5"
2. "Foundation 2.5m"
3. "Concrete C30/37"

→ Context lost, worse search ❌

4. Performance

Without chunking: 1 Document = 1 Embedding = 1 Search Result
With chunking: 1 Email = 3 Chunks = 3 Embeddings = 3 Search Results (need aggregation)

BUT: What if Attachment > 30,000 Characters?

Solution: Smart Truncating

csharp

```
private string PrepareAttachmentText(string fullText, string filename)
{
    if (fullText.Length <= 5000)
    {
        return fullText; // ✅ Fits completely
    }

    // Take first 2500 chars + last 2500 chars
    var start = fullText.Substring(0, 2500);
    var end = fullText.Substring(fullText.Length - 2500);

    return $"[{filename}]\n{start}\n\n... (truncated) ...\n\n{end}";
}
```

Why Start + End?

- **Start**: Title, Executive Summary, Introduction
- **End**: Conclusion, Signatures, Important Info
- **Middle**: Often repetitive (tables, details)

🔍 Search Examples

1. Simple Keyword Search

```
csharp

var results = await searchClient.SearchAsync<EmailSearchDocument>(
    searchText: "Construction Plan Hall 5",
    options: new SearchOptions
    {
        Filter = $"bu_guid eq '{currentUser.BuGuid}' and project_id eq {projectId}",
        OrderBy = { "created_at desc" },
        Select = { "email_guid", "subject", "sender_name", "created_at", "has_attachments" },
        Size = 20
    }
);
```

---

2. Semantic Search (AI-powered)

```csharp
var results = await searchClient.SearchAsync<EmailSearchDocument>(
    searchText: "foundation problems",
    options: new SearchOptions
    {
        Filter = $"bu_guid eq '{currentUser.BuGuid}' and project_id eq {projectId}",
        SemanticSearch = new SemanticSearchOptions
        {
            SemanticConfigurationName = "email-semantic-config",
            QueryCaption = new QueryCaption(QueryCaptionType.Extractive),
            QueryAnswer = new QueryAnswer(QueryAnswerType.Extractive)
        },
        Size = 20
    }
);
```

Also finds:

- "issues with base plate"
- "cracks in foundation"
- "structural problems underground"

---

3. Vector Search (Similarity)

```csharp
```

```
// Generate query embedding
var queryEmbedding = await openAI.GenerateEmbeddingAsync("Construction Plan Hall 5");

var results = await searchClient.SearchAsync<EmailSearchDocument>(
    searchText: null,
    options: new SearchOptions
    {
        Filter = $"bu_guid eq '{currentUser.BuGuid}' and project_id eq {projectId}",
        VectorSearch = new VectorSearchOptions
        {
            Queries =
            {
                new VectorizedQuery(queryEmbedding)
                {
                    KNearestNeighborsCount = 20,
                    Fields = { "content_vector" }
                }
            },
        Size = 20
    }
);
```

---

4. Hybrid Search (RECOMMENDED)

csharp

```
var queryEmbedding = await openAI.GenerateEmbeddingAsync(searchText);

var results = await searchClient.SearchAsync<EmailSearchDocument>(
    searchText: "Construction Plan Hall 5",  // Keyword
    options: new SearchOptions
    {
        Filter = $"bu_guid eq '{currentUser.BuGuid}' and project_id eq {projectId}",

        // Keyword Search
        QueryType = SearchQueryType.Full,

        // Vector Search
        VectorSearch = new VectorSearchOptions
        {
            Queries =
            {
                new VectorizedQuery(queryEmbedding)
                {
                    KNearestNeighborsCount = 50,
                    Fields = { "content_vector" }
                }
            }
        },

        // Semantic Ranking
        SemantiSearch = new SemanticSearchOptions
        {
            SemanticConfigurationName = "email-semantic-config"
        },

        Size = 20
    }
);
```

Combines:

- ✅ Keyword Match (Construction, Plan, Hall, 5)
- ✅ Semantic Understanding (building plans, construction designs)
- ✅ Vector Similarity (similar documents)

→ Best Results! 🎯

---

5. Attachment Search

```csharp
// Search for emails with specific attachment
var results = await searchClient.SearchAsync<EmailSearchDocument>(
    searchText: "Construction_Plan_Hall5_v2.pdf",
    options: new SearchOptions
    {
        Filter = $"bu_guid eq '{currentUser.BuGuid}' and project_id eq {projectId}",
        SearchFields = { "attachment_names" },
        Size = 20
    }
);
```

---

6. Faceted Search

```csharp
```

```
var results = await searchClient.SearchAsync<EmailSearchDocument>(
    searchText: "*",
    options: new SearchOptions
    {
        Filter = $"bu_guid eq '{currentUser.BuGuid}' and project_id eq {projectId}",
        Facets =
        {
            "sender_email,count:10",
            "attachment_types,count:5",
            "keypoint_id,count:20",
            "created_at,interval:month"
        },
        Size = 0  // Only get facets, no documents
    }
);

// Results:
// Facets["sender_email"]:
//   - ibrahim.azim@semyou.com: 45 emails
//   - volker.jahns@semyou.com: 32 emails
// Facets["attachment_types"]:
//   - application/pdf: 87 attachments
//   - application/vnd...xlsx: 23 attachments
```

💰 Cost Estimation

Scenario: 100 Emails/Day, 3 Attachments/Email

Monthly:
- 3,000 Emails
- 9,000 Attachments
- ~360 Image-PDFs for OCR (10%)

Azure Document Intelligence:
360 PDFs × 10 pages = 3,600 pages
3,600 ÷ 1,000 × €1.50 = €5.40

OpenAI Embeddings:
3,000 Emails × 2,000 chars avg = 6M chars ≈ 1.5M tokens
1.5M ÷ 1M × €0.10 = €0.15

Azure AI Search (Basic tier):
  €75/month (flat fee)

TOTAL: €80.55/month
Per Email: €0.027 (2.7 cents!)

🎯 Summary

Index Design Decisions:

Decision	Reasoning
No Chunking	Emails are small enough (<30K chars)
body_text instead of body_html	Plain text better for search, HTML redundant
content_combined	Single field for embedding = better context
attachment_texts Collection	All attachments searchable in one document
German Analyzer	Optimized for German construction texts
Hybrid Search	Keyword + Semantic + Vector = best results
MANDATORY Filters	bu_guid + project_id = Row-Level Security

Processing Pipeline:

```
Mailgun → Azure Function
  ├─> Parse & Validate
  ├─> Extract Attachment Texts (parallel)
  │   ├─> TXT/DOCX/PPTX → Free libraries
  │   └─> PDF → PdfPig or Azure DI
  ├─> Combine Content
  ├─> Generate Embedding (OpenAI)
  └─> Index in Azure Search
```

Performance:

- **Email Processing**: < 5 sec (without OCR)

- **With OCR**: < 15 sec (90% under 30 sec)
- **Search Query**: < 200ms (Hybrid Search)

Cost:

- **~€80/month** for 100 Emails/Day
- **€0.027 per Email** (2.7 cents!)

📋 Implementation Checklist

- [ ] Create Azure AI Search service (or use existing: pcpx-search.search.windows.net)
- [ ] Create index using EmailInboundSearchIndex.json
- [ ] Add NuGet packages: Azure.Search.Documents, Azure.AI.OpenAI
- [ ] Implement AttachmentTextExtractor
- [ ] Implement EmailSearchService
- [ ] Extend MailgunInbound Function
- [ ] Test with real emails
- [ ] Monitor costs and performance

Ready for implementation! 🚀