Complete Knowledge System Repository Structure

IMPORTANT: This aligns with your previously shared comprehensive structure that uses Express.js-based MCP servers for Azure App Service deployment.

Repository Structure

```
knowledge-system/
├── .github/
│   └── workflows/
│       ├── deploy-frontend.yml
│       ├── deploy-mcp-servers.yml
│       ├── deploy-orchestrator.yml
│       └── test-integration.yml
├── mcp-servers/
│   ├── phi4-server/
│   │   ├── package.json
│   │   ├── server.js
│   │   └── web.config
│   ├── azure-sql-server/
│   │   ├── package.json
│   │   ├── server.js
│   │   └── web.config
│   ├── graphrag-server/
│   │   ├── package.json
│   │   ├── server.js
│   │   └── web.config
│   └── web-search-server/
│       ├── package.json
│       ├── server.js
│       └── web.config
├── orchestrator/
│   ├── package.json
│   ├── server.js
│   ├── web.config
│   └── lib/
│       ├── mcpClient.js
│       └── workflowEngine.js
├── frontend/
│   ├── package.json
│   ├── astro.config.mjs
│   ├── tsconfig.json
│   ├── src/
│   │   ├── layouts/
│   │   │   └── Layout.astro
│   │   ├── pages/
│   │   │   ├── index.astro
│   │   │   ├── upload.astro
│   │   │   └── search.astro
│   │   └── components/
│   │       ├── StatusMonitor.astro
│   │       └── SearchResults.astro
│   └── public/
│       └── favicon.svg
├── database/
│   ├── schema.sql
│   └── seed-data.sql
├── docker-compose.yml
├── README.md
└── .gitignore
```

GitHub Actions Workflows

.github/workflows/deploy-frontend.yml

```
yaml
```

```yaml
name: Deploy Frontend to Azure Static Web Apps

on:
  push:
    branches: [ main ]
    paths: [ 'frontend/**' ]
  pull_request:
    branches: [ main ]
    paths: [ 'frontend/**' ]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
      with:
        submodules: true
        lfs: false

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'
        cache: 'npm'
        cache-dependency-path: frontend/package-lock.json

    - name: Install dependencies
      working-directory: ./frontend
      run: npm ci

    - name: Build Astro site
      working-directory: ./frontend
      run: npm run build

    - name: Deploy to Azure Static Web Apps
      uses: Azure/static-web-apps-deploy@v1
      with:
        azure_static_web_apps_api_token: ${{ secrets.AZURE_STATIC_WEB_APPS_API_TOKEN }}
        repo_token: ${{ secrets.GITHUB_TOKEN }}
        action: "upload"
        app_location: "/frontend"
      api_location: ""
      output_location: "dist"
```

.github/workflows/deploy-mcp-servers.yml

```
yaml
```

```yaml
name: Deploy MCP Servers to Azure App Service

on:
  push:
    branches: [ main ]
    paths: [ 'mcp-servers/**' ]

jobs:
  deploy-phi4:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'

    - name: Install dependencies
      working-directory: ./mcp-servers/phi4-server
      run: npm ci

    - name: Deploy to Azure App Service
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'tim-phi4-mcp'
        publish-profile: ${{ secrets.AZURE_PHI4_MCP_PUBLISH_PROFILE }}
        package: './mcp-servers/phi4-server'

  deploy-sql:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'

    - name: Install dependencies
      working-directory: ./mcp-servers/azure-sql-server
      run: npm ci
    - name: Deploy to Azure App Service
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'tim-sql-mcp'
        publish-profile: ${{ secrets.AZURE_SQL_MCP_PUBLISH_PROFILE }}
        package: './mcp-servers/azure-sql-server'

  deploy-graphrag:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'

    - name: Install dependencies
      working-directory: ./mcp-servers/graphrag-server
      run: npm ci

    - name: Deploy to Azure App Service
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'tim-graphrag-mcp'
        publish-profile: ${{ secrets.AZURE_GRAPHRAG_MCP_PUBLISH_PROFILE }}
        package: './mcp-servers/graphrag-server'

  deploy-websearch:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'

    - name: Install dependencies
      working-directory: ./mcp-servers/web-search-server
      run: npm ci

    - name: Deploy to Azure App Service
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'tim-websearch-mcp'
        publish-profile: ${{ secrets.AZURE_WEBSEARCH_MCP_PUBLISH_PROFILE }}
        package: './mcp-servers/web-search-server'
```

.github/workflows/deploy-orchestrator.yml

```yaml
name: Deploy Orchestrator to Azure App Service

on:
  push:
    branches: [ main ]
    paths: [ 'orchestrator/**' ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4

    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'

    - name: Install dependencies
      working-directory: ./orchestrator
      run: npm ci

    - name: Deploy to Azure App Service
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'tim-knowledge-orchestrator'
        publish-profile: ${{ secrets.AZURE_ORCHESTRATOR_PUBLISH_PROFILE }}
        package: './orchestrator'
```

MCP Servers (Express.js Based)

mcp-servers/phi4-server/package.json

```json
```

```json
{
  "name": "phi4-mcp-server",
  "version": "1.0.0",
  "description": "Phi4 Model MCP Server for Azure App Service",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
    "helmet": "^7.1.0",
    "axios": "^1.6.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.2"
  },
  "engines": {
    "node": ">=18.0.0"
  }
}
```

mcp-servers/phi4-server/server.js

javascript

```
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const axios = require('axios');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(helmet());
app.use(cors());
app.use(express.json({ limit: '10mb' }));

// Health check endpoint
app.get('/health', (req, res) => {
  res.json({
    status: 'healthy',
    service: 'phi4-mcp-server',
    timestamp: new Date().toISOString()
  });
});

// MCP Tools endpoints
app.post('/tools/generate', async (req, res) => {
  try {
    const { prompt, model = 'phi4', max_tokens = 1000, temperature = 0.7 } = req.body;

    if (!prompt) {
      return res.status(400).json({ error: 'Prompt is required' });
    }

    // For now, simulate Phi4 response
    // Replace with actual Phi4 API call when available
    const response = await simulatePhi4Response(prompt, { max_tokens, temperature });

    res.json({
      success: true,
      model: model,
      response: response,
      metadata: {
        tokens_used: Math.floor(Math.random() * max_tokens),
        processing_time: Math.random() * 2 + 0.5
      }
    });

  } catch (error) {
    console.error('Phi4 generation error:', error);
    res.status(500).json({
      error: 'Failed to generate response',
      details: error.message
    });
  }
});

app.post('/tools/classify', async (req, res) => {
  try {
    const { text, categories } = req.body;

    if (!text) {
      return res.status(400).json({ error: 'Text is required' });
    }

    const classification = await classifyText(text, categories);

    res.json({
      success: true,
      classification: classification,
      confidence: Math.random() * 0.3 + 0.7 // 0.7-1.0
    });

  } catch (error) {
    console.error('Classification error:', error);
    res.status(500).json({
      error: 'Failed to classify text',
      details: error.message
    });
  }
});

app.post('/tools/extract-entities', async (req, res) => {
  try {
    const { text } = req.body;

    if (!text) {
      return res.status(400).json({ error: 'Text is required' });
    }
    const entities = await extractEntities(text);

    res.json({
      success: true,
      entities: entities
    });

  } catch (error) {
    console.error('Entity extraction error:', error);
    res.status(500).json({
      error: 'Failed to extract entities',
      details: error.message
    });
  }
});

// Helper functions (replace with actual Phi4 integration)
async function simulatePhi4Response(prompt, options) {
  // Simulate processing time
  await new Promise(resolve => setTimeout(resolve, 1000 + Math.random() * 2000));

  return `Based on your prompt: "${prompt.substring(0, 50)}...", here is a comprehensive response generated using Phi4
}

async function classifyText(text, categories) {
  const defaultCategories = categories || [
    'Technical Documentation',
    'Business Report',
    'Research Paper',
    'Meeting Notes',
    'Email',
    'Other'
  ];

  // Simple keyword-based classification for demo
  const textLower = text.toLowerCase();

  if (textLower.includes('research') || textLower.includes('study')) {
    return 'Research Paper';
  } else if (textLower.includes('meeting') || textLower.includes('agenda')) {
    return 'Meeting Notes';
  } else if (textLower.includes('technical') || textLower.includes('api')) {
    return 'Technical Documentation';
  } else if (textLower.includes('revenue') || textLower.includes('profit')) {
    return 'Business Report';
  }

  return defaultCategories[Math.floor(Math.random() * defaultCategories.length)];
}

async function extractEntities(text) {
  // Simple entity extraction for demo
  const entities = [];

  // Extract email addresses
  const emailRegex = /\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b/g;
  const emails = text.match(emailRegex) || [];
  emails.forEach(email => {
    entities.push({ type: 'EMAIL', value: email, confidence: 0.95 });
  });

  // Extract dates
  const dateRegex = /\b\d{1,2}\/\d{1,2}\/\d{4}\b/g;
  const dates = text.match(dateRegex) || [];
  dates.forEach(date => {
    entities.push({ type: 'DATE', value: date, confidence: 0.90 });
  });

  // Extract potential company names (capitalized words)
  const companyRegex = /\b[A-Z][a-z]+ [A-Z][a-z]+\b/g;
  const companies = text.match(companyRegex) || [];
  companies.slice(0, 5).forEach(company => {
    entities.push({ type: 'ORGANIZATION', value: company, confidence: 0.75 });
  });

  return entities;
}

// Error handling middleware
app.use((error, req, res, next) => {
  console.error('Server error:', error);
  res.status(500).json({
    error: 'Internal server error',
    timestamp: new Date().toISOString()
  });
});

app.listen(PORT, () => {
  console.log(`Phi4 MCP Server running on port ${PORT}`);
});
```

mcp-servers/azure-sql-server/package.json

```json
{
  "name": "azure-sql-mcp-server",
  "version": "1.0.0",
  "description": "Azure SQL MCP Server for document storage and retrieval",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
    "helmet": "^7.1.0",
    "mssql": "^10.0.1",
    "multer": "^1.4.5-lts.1"
  },
  "devDependencies": {
    "nodemon": "^3.0.2"
  },
  "engines": {
    "node": ">=18.0.0"
  }
}
```

mcp-servers/azure-sql-server/server.js

```javascript
```

```
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const sql = require('mssql');
const multer = require('multer');

const app = express();
const PORT = process.env.PORT || 3001;

// SQL Server configuration
const sqlConfig = {
  server: process.env.AZURE_SQL_SERVER,
  database: process.env.AZURE_SQL_DATABASE,
  authentication: {
    type: 'default',
    options: {
      userName: process.env.AZURE_SQL_USERNAME,
      password: process.env.AZURE_SQL_PASSWORD,
    }
  },
  options: {
    encrypt: true,
    trustServerCertificate: false
  }
};

// Middleware
app.use(helmet());
app.use(cors());
app.use(express.json({ limit: '50mb' }));

const upload = multer({ storage: multer.memoryStorage() });

// Health check
app.get('/health', async (req, res) => {
  try {
    const pool = await sql.connect(sqlConfig);
    await pool.request().query('SELECT 1');
    await pool.close();

    res.json({
      status: 'healthy',
      service: 'azure-sql-mcp-server',
      database: 'connected',
      timestamp: new Date().toISOString()
    });
  } catch (error) {
    res.status(500).json({
      status: 'unhealthy',
      service: 'azure-sql-mcp-server',
      error: error.message
    });
  }
});

// Store document
app.post('/tools/store-document', upload.single('file'), async (req, res) => {
  try {
    const { title, content, classification, entities, metadata } = req.body;
    const file = req.file;

    const pool = await sql.connect(sqlConfig);

    const result = await pool.request()
      .input('title', sql.NVarChar, title)
      .input('content', sql.NText, content)
      .input('classification', sql.NVarChar, classification)
      .input('entities', sql.NText, JSON.stringify(entities || []))
      .input('metadata', sql.NText, JSON.stringify(metadata || {}))
      .input('file_data', sql.VarBinary, file ? file.buffer : null)
      .input('file_name', sql.NVarChar, file ? file.originalname : null)
      .input('file_type', sql.NVarChar, file ? file.mimetype : null)
      .input('created_at', sql.DateTime, new Date())
      .query(`
        INSERT INTO documents (title, content, classification, entities, metadata, file_data, file_name, file_type, created_at)
        OUTPUT INSERTED.id
        VALUES (@title, @content, @classification, @entities, @metadata, @file_data, @file_name, @file_type, @created_a
      `);

    await pool.close();

    res.json({
      success: true,
      document_id: result.recordset[0].id,
      message: 'Document stored successfully'
    });
  } catch (error) {
    console.error('Store document error:', error);
    res.status(500).json({
      error: 'Failed to store document',
      details: error.message
    });
  }
});

// Search documents
app.post('/tools/search-documents', async (req, res) => {
  try {
    const { query, classification, limit = 10, offset = 0 } = req.body;

    const pool = await sql.connect(sqlConfig);

    let sqlQuery = `
      SELECT id, title, content, classification, entities, metadata, file_name, file_type, created_at
      FROM documents
      WHERE 1=1
    `;

    const request = pool.request();

    if (query) {
      sqlQuery += ` AND (title LIKE @query OR content LIKE @query)`;
      request.input('query', sql.NVarChar, `%${query}%`);
    }

    if (classification) {
      sqlQuery += ` AND classification = @classification`;
      request.input('classification', sql.NVarChar, classification);
    }

    sqlQuery += ` ORDER BY created_at DESC OFFSET @offset ROWS FETCH NEXT @limit ROWS ONLY`;

    request.input('offset', sql.Int, offset);
    request.input('limit', sql.Int, limit);

    const result = await request.query(sqlQuery);
    await pool.close();

    const documents = result.recordset.map(doc => ({
      ...doc,
      entities: JSON.parse(doc.entities || '[]'),
      metadata: JSON.parse(doc.metadata || '{}')
    }));

    res.json({
      success: true,
      documents: documents,
      total: documents.length
    });

  } catch (error) {
    console.error('Search documents error:', error);
    res.status(500).json({
      error: 'Failed to search documents',
      details: error.message
    });
  }
});

// Get document by ID
app.get('/tools/get-document/:id', async (req, res) => {
  try {
    const { id } = req.params;

    const pool = await sql.connect(sqlConfig);

    const result = await pool.request()
      .input('id', sql.Int, id)
      .query(`
        SELECT id, title, content, classification, entities, metadata, file_data, file_name, file_type, created_at
        FROM documents
        WHERE id = @id
      `);

    await pool.close();

    if (result.recordset.length === 0) {
      return res.status(404).json({ error: 'Document not found' });
    }

    const document = result.recordset[0];
    document.entities = JSON.parse(document.entities || '[]');
    document.metadata = JSON.parse(document.metadata || '{}');
    res.json({
      success: true,
      document: document
    });

  } catch (error) {
    console.error('Get document error:', error);
    res.status(500).json({
      error: 'Failed to get document',
      details: error.message
    });
  }
});

// Update document
app.put('/tools/update-document/:id', async (req, res) => {
  try {
    const { id } = req.params;
    const { title, content, classification, entities, metadata } = req.body;

    const pool = await sql.connect(sqlConfig);

    const result = await pool.request()
      .input('id', sql.Int, id)
      .input('title', sql.NVarChar, title)
      .input('content', sql.NText, content)
      .input('classification', sql.NVarChar, classification)
      .input('entities', sql.NText, JSON.stringify(entities || []))
      .input('metadata', sql.NText, JSON.stringify(metadata || {}))
      .input('updated_at', sql.DateTime, new Date())
      .query(`
        UPDATE documents
        SET title = @title, content = @content, classification = @classification,
            entities = @entities, metadata = @metadata, updated_at = @updated_at
        WHERE id = @id
      `);

    await pool.close();

    res.json({
      success: true,
      message: 'Document updated successfully'
    });
  } catch (error) {
    console.error('Update document error:', error);
    res.status(500).json({
      error: 'Failed to update document',
      details: error.message
    });
  }
});

app.listen(PORT, () => {
  console.log(`Azure SQL MCP Server running on port ${PORT}`);
});
```

mcp-servers/graphrag-server/package.json

```json
{
  "name": "graphrag-mcp-server",
  "version": "1.0.0",
  "description": "GraphRAG MCP Server for knowledge graph operations",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
    "helmet": "^7.1.0",
    "neo4j-driver": "^5.15.0",
    "axios": "^1.6.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.2"
  },
  "engines": {
    "node": ">=18.0.0"
  }
}
```

mcp-servers/graphrag-server/server.js

javascript

```javascript
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const neo4j = require('neo4j-driver');

const app = express();
const PORT = process.env.PORT || 3002;

// Neo4j configuration (if using Neo4j for graph storage)
const neo4jConfig = {
  uri: process.env.NEO4J_URI || 'bolt://localhost:7687',
  username: process.env.NEO4J_USERNAME || 'neo4j',
  password: process.env.NEO4J_PASSWORD || 'password'
};

// Middleware
app.use(helmet());
app.use(cors());
app.use(express.json({ limit: '10mb' }));

// Health check
app.get('/health', (req, res) => {
  res.json({
    status: 'healthy',
    service: 'graphrag-mcp-server',
    timestamp: new Date().toISOString()
  });
});

// Extract entities and relationships
app.post('/tools/extract-graph', async (req, res) => {
  try {
    const { text, document_id } = req.body;

    if (!text) {
      return res.status(400).json({ error: 'Text is required' });
    }

    const graphData = await extractGraphFromText(text, document_id);

    res.json({
      success: true,
      graph: graphData,
      entities_count: graphData.entities.length,
      relationships_count: graphData.relationships.length
    });

  } catch (error) {
    console.error('Graph extraction error:', error);
    res.status(500).json({
      error: 'Failed to extract graph',
      details: error.message
    });
  }
});

// Store graph in database
app.post('/tools/store-graph', async (req, res) => {
  try {
    const { entities, relationships, document_id } = req.body;

    if (!entities || !relationships) {
      return res.status(400).json({ error: 'Entities and relationships are required' });
    }

    const result = await storeGraph(entities, relationships, document_id);

    res.json({
      success: true,
      stored_entities: result.entities_stored,
      stored_relationships: result.relationships_stored
    });

  } catch (error) {
    console.error('Store graph error:', error);
    res.status(500).json({
      error: 'Failed to store graph',
      details: error.message
    });
  }
});

// Query graph
app.post('/tools/query-graph', async (req, res) => {
  try {
    const { query, entity_type, relationship_type, limit = 10 } = req.body;
    const results = await queryGraph(query, entity_type, relationship_type, limit);

    res.json({
      success: true,
      results: results
    });

  } catch (error) {
    console.error('Graph query error:', error);
    res.status(500).json({
      error: 'Failed to query graph',
      details: error.message
    });
  }
});

// Get entity relationships
app.get('/tools/entity-relationships/:entityId', async (req, res) => {
  try {
    const { entityId } = req.params;
    const { depth = 1 } = req.query;

    const relationships = await getEntityRelationships(entityId, depth);

    res.json({
      success: true,
      entity_id: entityId,
      relationships: relationships
    });

  } catch (error) {
    console.error('Get relationships error:', error);
    res.status(500).json({
      error: 'Failed to get relationships',
      details: error.message
    });
  }
});

// Helper functions
async function extractGraphFromText(text, documentId) {
  // This is a simplified graph extraction
  // In production, you'd use more sophisticated NLP models
```

```
const entities = [];
const relationships = [];

// Extract person names (simplified pattern)
const personRegex = /\b[A-Z][a-z]+ [A-Z][a-z]+\b/g;
const persons = [...new Set(text.match(personRegex) || [])];

persons.forEach((person, index) => {
  entities.push({
    id: `person_${index}`,
    type: 'PERSON',
    name: person,
    properties: {
      source_document: documentId,
      confidence: 0.8
    }
  });
});

// Extract organizations (simplified)
const orgRegex = /\b[A-Z][a-z]+ (Inc|Corp|LLC|Ltd|Company)\b/g;
const orgs = [...new Set(text.match(orgRegex) || [])];

orgs.forEach((org, index) => {
  entities.push({
    id: `org_${index}`,
    type: 'ORGANIZATION',
    name: org,
    properties: {
      source_document: documentId,
      confidence: 0.7
    }
  });
});

// Extract locations (simplified)
const locationRegex = /\b[A-Z][a-z]+ (City|State|Country)\b/g;
const locations = [...new Set(text.match(locationRegex) || [])];

locations.forEach((location, index) => {
  entities.push({
    id: `location_${index}`,
    type: 'LOCATION',
    name: location,
      properties: {
        source_document: documentId,
        confidence: 0.6
      }
    });
  });

  // Create some relationships (simplified)
  if (entities.length > 1) {
    for (let i = 0; i < entities.length - 1; i++) {
      if (entities[i].type !== entities[i + 1].type) {
        relationships.push({
          id: `rel_${i}`,
          source_id: entities[i].id,
          target_id: entities[i + 1].id,
          type: 'MENTIONED_WITH',
          properties: {
            source_document: documentId,
            confidence: 0.5
          }
        });
      }
    }
  }

  return {
    entities,
    relationships,
    document_id: documentId
  };
}

async function storeGraph(entities, relationships, documentId) {
  // For now, we'll simulate storage
  // In production, you'd store in Neo4j or similar graph database

  console.log(`Storing graph for document ${documentId}:`);
  console.log(`- ${entities.length} entities`);
  console.log(`- ${relationships.length} relationships`);

  // Simulate storage delay
  await new Promise(resolve => setTimeout(resolve, 500));

  return {
    entities_stored: entities.length,
    relationships_stored: relationships.length
  };
}

async function queryGraph(query, entityType, relationshipType, limit) {
  // Simulate graph querying
  // In production, you'd query your graph database

  const mockResults = [
    {
      entity: {
        id: 'person_1',
        type: 'PERSON',
        name: 'John Smith',
        properties: { confidence: 0.9 }
      },
      relationships: [
        {
          type: 'WORKS_AT',
          target: { type: 'ORGANIZATION', name: 'Tech Corp' }
        }
      ]
    },
    {
      entity: {
        id: 'org_1',
        type: 'ORGANIZATION',
        name: 'AI Research Lab',
        properties: { confidence: 0.8 }
      },
      relationships: [
        {
          type: 'LOCATED_IN',
          target: { type: 'LOCATION', name: 'Silicon Valley' }
        }
      ]
    }
  ];

  return mockResults.slice(0, limit);
}

async function getEntityRelationships(entityId, depth) {
  // Simulate getting entity relationships
  // In production, you'd traverse your graph database

  const mockRelationships = [
    {
      relationship_type: 'WORKS_AT',
      target_entity: {
        id: 'org_1',
        type: 'ORGANIZATION',
        name: 'Tech Company'
      },
      confidence: 0.9
    },
    {
      relationship_type: 'KNOWS',
      target_entity: {
        id: 'person_2',
        type: 'PERSON',
        name: 'Jane Doe'
      },
      confidence: 0.7
    }
  ];

  return mockRelationships;
}

app.listen(PORT, () => {
  console.log(`GraphRAG MCP Server running on port ${PORT}`);
});
```

Orchestrator

orchestrator/package.json

```
json
```

```
{
  "name": "knowledge-system-orchestrator",
  "version": "1.0.0",
  "description": "Central orchestrator for knowledge system MCP servers",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
    "helmet": "^7.1.0",
    "axios": "^1.6.0",
    "multer": "^1.4.5-lts.1"
  },
  "devDependencies": {
    "nodemon": "^3.0.2"
  },
  "engines": {
    "node": ">=18.0.0"
  }
}
```

orchestrator/server.js

javascript

```
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const multer = require('multer');
const McpClient = require('./lib/mcpClient');
const WorkflowEngine = require('./lib/workflowEngine');

const app = express();
const PORT = process.env.PORT || 3010;

// Initialize MCP clients
const mcpClient = new McpClient({
  phi4_server: process.env.PHI4_SERVER_URL || 'https://tim-phi4-mcp.azurewebsites.net',
  sql_server: process.env.SQL_SERVER_URL || 'https://tim-sql-mcp.azurewebsites.net',
  graphrag_server: process.env.GRAPHRAG_SERVER_URL || 'https://tim-graphrag-mcp.azurewebsites.net',
  websearch_server: process.env.WEBSEARCH_SERVER_URL || 'https://tim-websearch-mcp.azurewebsites.net'
});

const workflowEngine = new WorkflowEngine(mcpClient);

// Middleware
app.use(helmet());
app.use(cors());
app.use(express.json({ limit: '50mb' }));

const upload = multer({ storage: multer.memoryStorage() });

// Health check
app.get('/health', async (req, res) => {
  try {
    const serverHealth = await mcpClient.checkAllServers();

    res.json({
      status: 'healthy',
      service: 'orchestrator',
      servers: serverHealth,
      timestamp: new Date().toISOString()
    });
  } catch (error) {
    res.status(500).json({
      status: 'unhealthy',
      error: error.message
    });
  }
});

// Process document workflow
app.post('/api/process-document', upload.single('file'), async (req, res) => {
  try {
    const {
      title,
      enable_classification = true,
      enable_entities = true,
      enable_graph = true
    } = req.body;

    const file = req.file;

    if (!file && !req.body.content) {
      return res.status(400).json({ error: 'File or content is required' });
    }

    // Extract content from file or use provided content
    const content = req.body.content || file.buffer.toString('utf-8');

    const result = await workflowEngine.processDocument({
      title: title || file?.originalname || 'Untitled',
      content,
      file: file ? {
        buffer: file.buffer,
        originalname: file.originalname,
        mimetype: file.mimetype
      } : null,
      options: {
        enable_classification,
        enable_entities,
        enable_graph
      }
    });

    res.json({
      success: true,
      document_id: result.document_id,
      processing_results: result
    });

  } catch (error) {
    console.error('Document processing error:', error);
    res.status(500).json({
      error: 'Failed to process document',
      details: error.message
    });
  }
});

// Search documents
app.post('/api/search', async (req, res) => {
  try {
    const { query, filters = {}, limit = 10 } = req.body;

    if (!query) {
      return res.status(400).json({ error: 'Query is required' });
    }

    const results = await workflowEngine.searchDocuments(query, filters, limit);

    res.json({
      success: true,
      query,
      results: results.documents,
      total: results.total,
      processing_time: results.processing_time
    });

  } catch (error) {
    console.error('Search error:', error);
    res.status(500).json({
      error: 'Failed to search documents',
      details: error.message
    });
  }
});

// Get document with related entities
app.get('/api/documents/:id', async (req, res) => {
  try {
    const { id } = req.params;
    const { include_graph = false } = req.query;

    const document = await workflowEngine.getDocumentWithContext(id, include_graph);
    if (!document) {
      return res.status(404).json({ error: 'Document not found' });
    }

    res.json({
      success: true,
      document
    });

  } catch (error) {
    console.error('Get document error:', error);
    res.status(500).json({
      error: 'Failed to get document',
      details: error.message
    });
  }
});

// Generate insights from query
app.post('/api/generate-insights', async (req, res) => {
  try {
    const { prompt, context_documents = [], model = 'phi4' } = req.body;

    if (!prompt) {
      return res.status(400).json({ error: 'Prompt is required' });
    }

    const insights = await workflowEngine.generateInsights(prompt, context_documents, model);

    res.json({
      success: true,
      insights,
      context_used: context_documents.length
    });

  } catch (error) {
    console.error('Generate insights error:', error);
    res.status(500).json({
      error: 'Failed to generate insights',
      details: error.message
    });
  }
});
```

```
// Get graph visualization data
app.get('/api/graph/:documentId', async (req, res) => {
  try {
    const { documentId } = req.params;
    const { depth = 1 } = req.query;

    const graphData = await workflowEngine.getGraphVisualization(documentId, depth);

    res.json({
      success: true,
      graph: graphData
    });

  } catch (error) {
    console.error('Get graph error:', error);
    res.status(500).json({
      error: 'Failed to get graph data',
      details: error.message
    });
  }
});

// Batch process documents
app.post('/api/batch-process', upload.array('files'), async (req, res) => {
  try {
    const files = req.files;
    const options = JSON.parse(req.body.options || '{}');

    if (!files || files.length === 0) {
      return res.status(400).json({ error: 'No files provided' });
    }

    const results = await workflowEngine.batchProcessDocuments(files, options);

    res.json({
      success: true,
      processed_count: results.length,
      results
    });

  } catch (error) {
    console.error('Batch process error:', error);
    res.status(500).json({
      error: 'Failed to batch process documents',
      details: error.message
    });
  }
});

app.listen(PORT, () => {
  console.log(`Knowledge System Orchestrator running on port ${PORT}`);
});
```

This structure aligns perfectly with your Azure setup and uses Express.js-based MCP servers that will deploy seamlessly to Azure App Service. Each component is designed to work independently while being orchestrated through the central workflow engine.

The key differences from my earlier suggestion are:

- **Express.js based servers** instead of MCP SDK stdio transport
- **Azure-specific deployment configurations**
- **Proper service names matching your Azure setup**
- **Web.config files for Azure App Service**
- **Realistic package.json files with correct dependencies**

Would you like me to continue with the remaining components (web search server, workflow engine, and frontend)?