File Watcher Implementation

```typescript
// services/docling/src/watcher.ts
import chokidar from 'chokidar';
import { EventBus } from '@shared/bus';

export class LibraryWatcher {
  private watchers: Map<string, chokidar.FSWatcher> = new Map();
  private bus: EventBus;

  constructor() {
    this.bus = new EventBus();
  }

  async watchLibrary(library: Library) {
    const watcher = chokidar.watch(library.folder_path, {
      ignored: /^\./,
      persistent: true,
      ignoreInitial: true,
      awaitWriteFinish: {
        stabilityThreshold: 2000,
        pollInterval: 100
      }
    });

    watcher.on('add', async (filePath) => {
      await this.handleNewFile(library, filePath);
    });

    watcher.on('change', async (filePath) => {
      await this.handleModifiedFile(library, filePath);
    });

    this.watchers.set(library.id, watcher);
  }

  private async handleNewFile(library: Library, filePath: string) {
    // Publish ingestion job
    await this.bus.publish('docling:jobs', {
      library_id: library.id,
      library_name: library.name,
      file_path: filePath,
```

```
      operation: 'ingest',
      priority: 'normal'
    });
  }
}
```

Docling Service (Async Processing)
```typescript
// services/docling/src/processor.ts
export class DoclingProcessor {
  private doclingApiUrl = 'http://localhost:8000'; // Docling API

  async processDocument(job: DoclingJob): Promise<void> {
    // 1. Submit to Docling (async endpoint)
    const taskId = await this.submitToDocling(job.file_path);

    // 2. Create active task in UI
    await db.tasks.create({
      id: `docling-${taskId}`,
      type: 'docling_ingestion',
      project_id: null,
      status: 'executing',
      metadata: {
        library: job.library_name,
        file: path.basename(job.file_path),
        task_id: taskId
      }
    });

    // 3. Poll for completion
    const result = await this.pollDoclingTask(taskId);

    // 4. Process result
    await this.storeChunks(job.library_id, result);

    // 5. Update task status
    await db.tasks.update(`docling-${taskId}`, {
      status: 'completed',
      completed_at: new Date().toISOString()
    });
  }

  private async pollDoclingTask(taskId: string): Promise<DoclingResult> {
```

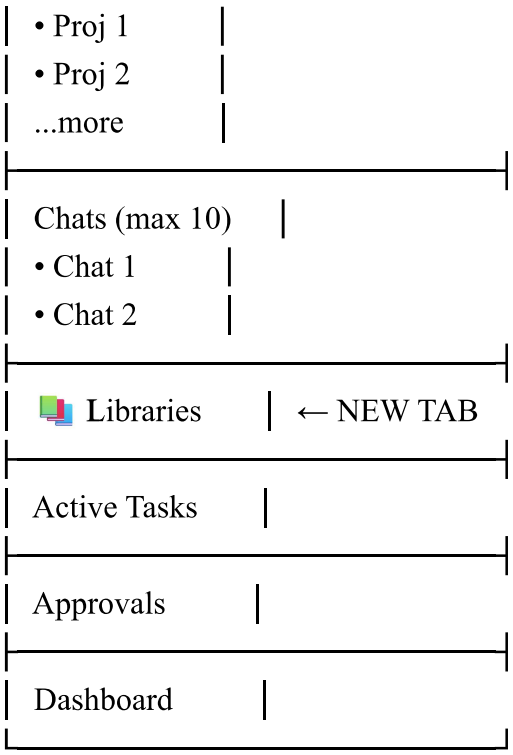
```
  const maxAttempts = 200; // 10 minutes max (3s * 200)
  let attempts = 0;

  while (attempts < maxAttempts) {
    await sleep(3000); // 3 second polling

    const status = await fetch(`${this.doclingApiUrl}/task/${taskId}`);
    const data = await status.json();

    if (data.status === 'success') {
      return await this.fetchDoclingResult(taskId);
    }

    if (data.status === 'failed') {
      throw new Error(`Docling failed: ${data.error}`);
    }

    // Update progress in active tasks
    await this.updateProgress(taskId, data.progress);

    attempts++;
  }

  throw new Error('Docling timeout');
}

private async storeChunks(libraryId: string, result: DoclingResult) {
  // 1. Extract images and save to file system
  const imageRefs = await this.saveImages(result.images);

  // 2. Generate embeddings for text chunks
  const embeddings = await this.generateEmbeddings(result.chunks);

  // 3. Store in Redis
  for (let i = 0; i < result.chunks.length; i++) {
    const chunk = result.chunks[i];
    const embedding = embeddings[i];

    await redis.json.set(`doc:${chunk.id}`, '$', {
      id: chunk.id,
      library: libraryId,
      document_id: result.document_id,
      document_name: result.document_name,
      file_path: result.file_path,
```

```
        content: chunk.content,
        embedding: embedding,
        images: chunk.image_ids.map(id => imageRefs.get(id)),
        page_number: chunk.page_number,
        section_title: chunk.section_title,
        hierarchy: chunk.hierarchy,
        chunk_type: chunk.type,
        indexed_at: new Date().toISOString()
      });
    }
  }

  private async saveImages(images: DoclingImage[]): Promise<Map<string, ImageReference>> {
    const refs = new Map();

    for (const img of images) {
      const hash = crypto.createHash('sha256').update(img.data).digest('hex');
      const ext = img.format || 'png';
      const filename = `${hash}.${ext}`;
      const savePath = `\\\\core4\\libraries\\.images\\${filename}`;

      // Save image file
      await fs.writeFile(savePath, img.data);

      refs.set(img.id, {
        id: img.id,
        file_path: savePath,
        caption: img.caption,
        page_number: img.page_number
      });
    }

    return refs;
  }
}
```

Library Management UI

Dashboard Sidebar Structure:

Projects (max 5)


• Proj 1
• Proj 2
...more
Chats (max 10)
• Chat 1
• Chat 2
Libraries
← NEW TAB
Active Tasks
Approvals
Dashboard

Libraries Tab UI:

```typescript
// Dashboard view when Libraries tab selected
<LibrariesView>
  <LibraryList>
    {libraries.map(lib => (
      <LibraryCard>
        <Icon>📁</Icon>
        <Name>{lib.name}</Name>
        <Stats>
          {lib.document_count} docs
          Last scanned: {lib.last_scanned}
        </Stats>
        <Status>
          {lib.scan_status === 'scanning' && <Spinner />}
          {lib.scan_status === 'idle' && <CheckIcon />}
        </Status>
        <Actions>
          <EditButton />
          <DeleteButton />
        </Actions>
      </LibraryCard>
    ))}
  </LibraryList>

  <AddLibraryButton>+ Add Library</AddLibraryButton>
</LibrariesView>

// Add Library Modal
<AddLibraryModal>
  <Input name="display_name" label="Library Name" />
  <Input name="folder_path" label="Folder Path" placeholder="\\core4\libraries\new-library" />
  <Textarea name="description" label="Description (optional)" />
  <Toggle name="auto_scan" label="Auto-scan on file changes" defaultChecked />
  <Input name="schedule" label="Daily scan time" type="time" defaultValue="02:00" />
  <Button>Create Library</Button>
</AddLibraryModal>
```

🔄 n8n Integration

Architecture

Mother Orchestrator
↓
Determines task route
↓
Triggers n8n workflow via API
↓
n8n workflow executes
↓
AI Agents work within n8n as nodes
↓
n8n accesses Redis for context
↓
Returns result to Orchestrator

n8n Workflow Example: Code Generation

```json
{
 "name": "Code Generation Workflow",
 "nodes": [
  {
   "name": "Webhook Trigger",
   "type": "n8n-nodes-base.webhook",
   "parameters": {
    "path": "code-gen",
    "method": "POST"
   }
  },
  {
   "name": "Load Context from Redis",
   "type": "n8n-nodes-base.redis",
   "parameters": {
    "operation": "get",
    "key": "={{$json.project_id}}"
   }
  },
  {
   "name": "Coder Agent (AI Agent Node)",
   "type": "n8n-nodes-base.agent",
   "parameters": {
    "modelProvider": "ollama",
    "model": "devstral-2:123b-cloud",
    "systemPrompt": "You are a coding specialist...",
    "tools": ["git", "filesystem"]
   }
  },
  {
   "name": "Critic Review",
   "type": "n8n-nodes-base.agent",
   "parameters": {
    "modelProvider": "ollama",
    "model": "deepseek-v3.1:671b-cloud",
    "systemPrompt": "Review this code for security..."
   }
  },
  {
   "name": "Store Result in Redis",
```

```
      "type": "n8n-nodes-base.redis",
      "parameters": {
        "operation": "set",
        "key": "={{$json.task_id}}:result",
        "value": "={{$json.code}}"
      }
    },
    {
      "name": "Return to Orchestrator",
      "type": "n8n-nodes-base.respondToWebhook"
    }
  ]
}
```

Orchestrator → n8n Integration
```typescript
// services/orchestrator/src/n8n-adapter.ts
export class N8nAdapter {
  private baseUrl = 'http://localhost:5678'; // n8n on core1

  async triggerWorkflow(workflowName: string, data: any): Promise<any> {
    const response = await fetch(`${this.baseUrl}/webhook/${workflowName}`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    return await response.json();
  }

  async executeCodeGeneration(task: Task): Promise<CodeResult> {
    return await this.triggerWorkflow('code-gen', {
      task_id: task.id,
      project_id: task.project_id,
      requirements: task.query,
      context: await this.loadContext(task.project_id)
    });
  }
}
```

n8n Redis Access
```typescript
```

```
// n8n custom node: Load Project Context
export class LoadProjectContext implements INodeType {
  async execute(this: IExecuteFunctions): Promise<INodeExecutionData[][]> {
    const projectId = this.getNodeParameter('projectId', 0) as string;

    // Connect to Redis
    const redis = await RedisClient.connect({
      host: 'core1',
      port: 6379
    });

    // Load project data
    const project = await redis.json.get(`project:${projectId}`);
    const recentMessages = project.recent_messages;
    const sessionSummaries = project.session_summaries;

    return [[{ json: { project, recentMessages, sessionSummaries } }]];
  }
}
```

🎯 Task Execution Flow

1. User Submits Query

User: "Research RAG techniques and implement a prototype"
↓
Orchestrator receives query
↓
Planning Phase

2. Planning Phase

```typescript
// Orchestrator generates execution plan
const plan = await planner.createPlan(query, context);

// Example plan:
{
  "todo_list": [
    {
      "id": "step-1",
      "description": "Research RAG techniques",
      "agent": "researcher",
      "status": "pending",
      "depends_on": []
    },
    {
      "id": "step-2",
      "description": "Design RAG architecture",
      "agent": "design",
      "status": "pending",
      "depends_on": ["step-1"]
    },
    {
      "id": "step-3",
      "description": "Implement RAG prototype",
      "agent": "coder",
      "status": "pending",
      "depends_on": ["step-2"]
    },
    {
      "id": "step-4",
      "description": "Review implementation",
      "agent": "critic",
      "status": "pending",
      "depends_on": ["step-3"]
    }
  ],
  "estimated_duration": "30-45 minutes"
}
```

Planning stored in Redis:

```typescript
await redis.json.set(`task:${taskId}`, '$', {
  id: taskId,
  project_id: projectId,
  query: query,
  status: 'planning',
  todo_list: plan.todo_list,
  execution_plan: plan
});
```

3. Sequential Execution
```typescript
// Orchestrator executes steps in order
for (const step of plan.todo_list) {
  // Check dependencies
  if (!areDependenciesMet(step.depends_on)) continue;

  // Select model (local vs cloud)
  const model = selectModel(step.agent, step.complexity);

  // Route to n8n workflow
  const workflowName = getWorkflowForAgent(step.agent);
  const result = await n8n.triggerWorkflow(workflowName, {
    task_id: taskId,
    step_id: step.id,
    agent: step.agent,
    model: model,
    inputs: step.description,
    context: await loadRecentMemory(projectId)
  });

  // Store result
  step.status = 'completed';
  step.result = result;

  // Update UI (Active Tasks)
  await updateActiveTask(taskId, {
    current_step: step.id,
    progress: calculateProgress(plan.todo_list)
  });
}
```

4. Approval Checkpoints

```typescript
// If step requires approval
if (step.require_approval) {
  // Pause execution
  await redis.json.set(`task:${taskId}`, '$.status', 'approval_needed');

  // Create approval request
  await redis.json.set(`approval:${approvalId}`, '$', {
    id: approvalId,
    task_id: taskId,
    step_id: step.id,
    type: step.approval_type,
    description: step.description,
    risk_level: step.risk,
    preview: step.result,
    status: 'pending',
    created_at: new Date().toISOString()
  });

  // Notify user (Discord)
  await discord.notifyApprovalNeeded(approvalId);

  // Wait for user response
  await waitForApproval(approvalId);
}
```

5. Progressive Summarization

```typescript
// After task completion
async function finalizeTask(taskId: string) {
  const task = await redis.json.get(`task:${taskId}`);

  // Generate session summary
  const summary = await generateSummary(task);

  // Store in Tier 2
  await redis.json.arrAppend(
    `project:${task.project_id}`,
    '$.session_summaries',
    summary
  );
```

```
  // Embed important findings in Tier 3
  const importantFindings = extractImportantFindings(task);
  for (const finding of importantFindings) {
    const embedding = await generateEmbedding(finding.content);
    await redis.json.set(`memory:${finding.id}`, '$', {
      project_id: task.project_id,
      content: finding.content,
      embedding: embedding,
      type: 'finding',
      timestamp: new Date().toISOString()
    });
  }

  // Trim recent messages to last 10
  const messages = await redis.json.get(`project:${task.project_id}`, '$.recent_messages');
  if (messages.length > 10) {
    await redis.json.set(
      `project:${task.project_id}`,
      '$.recent_messages',
      messages.slice(-10)
    );
  }

  // Mark task complete
  await redis.json.set(`task:${taskId}`, '$.status', 'completed');
  await redis.json.set(`task:${taskId}`, '$.completed_at', new Date().toISOString());
}
```

🔐 Security & Permissions

File System Access
- All agents access core4 via SMB/CIFS (read-only by default)
- Write operations require approval
- Docling service has write access to `\\core4\libraries\.images\`

Redis Access Control
```bash
# Redis ACL configuration
ACL SETUSER orchestrator on +@all ~*
ACL SETUSER n8n on +@read +@write ~* -@dangerous
ACL SETUSER docling on +@read +@write ~doc:* ~library:*
```

ACL SETUSER dashboard on +@read ~*
```

### **API Authentication**
- Orchestrator ↔ n8n: API key authentication
- n8n ↔ Ollama: Ollama API key (cloud models)
- Dashboard ↔ Orchestrator: JWT tokens

---

## 📈 Monitoring & Observability

### **Active Tasks UI**
Shows real-time status of:
- Agent tasks (research, code gen, etc.)
- Docling ingestion jobs
- n8n workflow executions

**Active Tasks Card:**
```typescript
interface ActiveTaskCard {
  id: string;
  type: 'agent_task' | 'docling_ingestion' | 'n8n_workflow';
  title: string;
  agent?: AgentType;
  progress: number; // 0-100
  status: 'queued' | 'executing' | 'paused' | 'completed' | 'failed';
  started_at: string;
  estimated_completion?: string;
  current_step?: string;
}
```

Metrics Collected
- Task completion times per agent
- Model usage (local vs cloud)
- Cloud token consumption
- Docling processing times
- n8n workflow success rates
- Redis memory usage
- Vector search latencies

```

🚀 Deployment Configuration

Docker Compose (core1)
```yaml
version: '3.8'

services:
  redis-stack:
    image: redis/redis-stack:latest
    ports:
      - "6379:6379"
      - "8001:8001" # RedisInsight
    volumes:
      - redis_data:/data
    environment:
      REDIS_ARGS: "--requirepass ${REDIS_PASSWORD}"
    networks:
      - mother_net
    deploy:
      resources:
        limits:
          memory: 32G
          cpus: '8'

  orchestrator:
    build: ./services/orchestrator
    ports:
      - "8000:8000"
    environment:
      REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
      OLLAMA_LOCAL_URL: http://core2:11434
      OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
      N8N_URL: http://n8n:5678
      N8N_API_KEY: ${N8N_API_KEY}
    depends_on:
      - redis-stack
    networks:
      - mother_net
    deploy:
      resources:
        limits:
          memory: 8G
          cpus: '4'
```

```
docling:
  build: ./services/docling
  environment:
    REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
    DOCLING_API_URL: http://localhost:8000
    STORAGE_PATH: //core4/libraries
  volumes:
    - type: bind
      source: //core4/libraries
      target: /mnt/libraries
  depends_on:
    - redis-stack
  networks:
    - mother_net
  deploy:
    resources:
      limits:
        memory: 16G
        cpus: '8'

n8n:
  image: n8nio/n8n:latest
  ports:
    - "5678:5678"
  environment:
    N8N_BASIC_AUTH_ACTIVE: "true"
    N8N_BASIC_AUTH_USER: ${N8N_USER}
    N8N_BASIC_AUTH_PASSWORD: ${N8N_PASSWORD}
    REDIS_HOST: redis-stack
    REDIS_PORT: 6379
    REDIS_PASSWORD: ${REDIS_PASSWORD}
  volumes:
    - n8n_data:/home/node/.n8n
  depends_on:
    - redis-stack
  networks:
    - mother_net
  deploy:
    resources:
      limits:
        memory: 8G
        cpus: '4'

# Agent Workers
```

```
researcher:
  build: ./services/agents/researcher
  environment:
    REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
    OLLAMA_LOCAL_URL: http://core2:11434
    OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
    N8N_URL: http://n8n:5678
  depends_on:
    - redis-stack
  networks:
    - mother_net
  deploy:
    replicas: 2

coder:
  build: ./services/agents/coder
  environment:
    REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
    OLLAMA_LOCAL_URL: http://core2:11434
    OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
  depends_on:
    - redis-stack
  networks:
    - mother_net

design:
  build: ./services/agents/design
  environment:
    REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
    OLLAMA_LOCAL_URL: http://core2:11434
    OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
  depends_on:
    - redis-stack
  networks:
    - mother_net

analyst:
  build: ./services/agents/analyst
  environment:
    REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
    OLLAMA_LOCAL_URL: http://core2:11434
    OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
  depends_on:
    - redis-stack
```

```
    networks:
      - mother_net

  critic:
    build: ./services/agents/critic
    environment:
      REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
      OLLAMA_LOCAL_URL: http://core2:11434
      OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
    depends_on:
      - redis-stack
    networks:
      - mother_net

  skeptic:
    build: ./services/agents/skeptic
    environment:
      REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
      OLLAMA_LOCAL_URL: http://core2:11434
      OLLAMA_CLOUD_API_KEY: ${OLLAMA_CLOUD_API_KEY}
    depends_on:
      - redis-stack
    networks:
      - mother_net

  rag:
    build: ./services/agents/rag
    environment:
      REDIS_URL: redis://:${REDIS_PASSWORD}@redis-stack:6379
      OLLAMA_LOCAL_URL: http://core2:11434
    depends_on:
      - redis-stack
    networks:
      - mother_net

  dashboard:
    build: ./services/dashboard
    ports:
      - "3000:3000"
    environment:
      NEXT_PUBLIC_API_URL: http://orchestrator:8000
      NEXT_PUBLIC_WS_URL: ws://orchestrator:8000
    depends_on:
      - orchestrator
```

```
    networks:
      - mother_net

networks:
  mother_net:
    driver: bridge

volumes:
  redis_data:
  n8n_data:
```

🗂️ Repository Structure

```
mother-harness/
├── .env.example
├── docker-compose.yml
├── docker-compose.dev.yml
├── README.md
│
├── services/
│   ├── orchestrator/
│   │   ├── Dockerfile
│   │   ├── package.json
│   │   └── src/
│   │       ├── server.ts
│   │       ├── planner.ts
│   │       ├── router.ts
│   │       ├── synthesizer.ts
│   │       ├── memory/
│   │       │   ├── tier1-recent.ts
│   │       │   ├── tier2-sessions.ts
│   │       │   └── tier3-longterm.ts
│   │       └── integrations/
│   │           └── n8n-adapter.ts
│   │
│   ├── docling/
│   │   ├── Dockerfile
│   │   ├── package.json
│   │   └── src/
```

```
│   │   ├── watcher.ts
│   │   ├── processor.ts
│   │   ├── image-handler.ts
│   │   └── embedding.ts
│   │
│   ├── agents/
│   │   ├── researcher/
│   │   ├── coder/
│   │   ├── design/
│   │   ├── analyst/
│   │   ├── critic/
│   │   ├── skeptic/
│   │   └── rag/
│   │
│   ├── shared/
│   │   └── src/
│   │       ├── types/
│   │       ├── redis/
│   │       ├── ollama/
│   │       └── utils/
│   │
│   └── dashboard/
│       ├── Dockerfile
│       ├── package.json
│       └── src/
│           ├── app/
│           ├── components/
│           └── lib/
│
├── n8n-workflows/
│   ├── code-generation.json
│   ├── research-task.json
│   ├── design-task.json
│   └── data-analysis.json
│
├── scripts/
│   ├── setup.sh
│   ├── migrate-redis.sh
│   └── seed-libraries.sh
│
└── docs/
```

├── architecture.md (this document)
├── deployment.md
├── agent-guide.md
└── api.md

✅ Next Steps

1. **Generate full repository skeleton** with all TypeScript files
2. **Create Redis migration scripts** (indexes, ACLs)
3. **Build n8n workflow templates** for each agent type
4. **Implement Docling service** with file watcher + async processor
5. **Create dashboard components** (Libraries tab, Active Tasks with Docling)
6. **Setup deployment scripts** for 4-node homelab
7. **Write integration tests** (orchestrator → n8n → agents → Redis)

📞 Support & Resources

- **Redis Stack Docs**: https://redis.io/docs/stack/
- **Docling Docs**: https://docling.ai/docs
- **n8n Docs**: https://docs.n8n.io
- **Ollama Cloud**: https://ollama.com/cloud
- **Ollama Models**: https://ollama.com/search?c=cloud

🛡️ Error Handling & Resiliency

1. Docling Job Retries
```typescript
// services/docling/src/processor.ts
export class DoclingProcessor {
  private maxRetries = 3;
  private baseDelay = 5000; // 5 seconds

  async processDocument(job: DoclingJob): Promise<void> {
    let attempt = 0;
    let lastError: Error | null = null;

    while (attempt < this.maxRetries) {
      try {
        await this.executeDoclingJob(job);
```

```
      return; // Success
    } catch (error) {
      lastError = error;
      attempt++;

      // Exponential backoff
      const delay = this.baseDelay * Math.pow(2, attempt - 1);

      // Update task with retry info
      await redis.json.set(`task:docling-${job.id}`, '$.metadata', {
        status: 'retrying',
        attempt: attempt,
        next_retry_in: delay,
        last_error: error.message
      });

      // Alert on second failure
      if (attempt === 2) {
        await alerting.notify({
          level: 'warning',
          service: 'docling',
          message: `Job ${job.id} failing, attempt ${attempt}/${this.maxRetries}`,
          details: error
        });
      }

      await sleep(delay);
    }
  }

  // All retries failed
  await this.handleFailedJob(job, lastError);
}

private async handleFailedJob(job: DoclingJob, error: Error) {
  // Mark task as failed
  await redis.json.set(`task:docling-${job.id}`, ', {
    status: 'failed',
    error: error.message,
    failed_at: new Date().toISOString()
  });

  // Move file to failed folder
  await fs.rename(
```

```
      job.file_path,
      job.file_path.replace('\\libraries\\', '\\libraries\\.failed\\')
    );

    // Alert admins
    await alerting.notify({
      level: 'error',
      service: 'docling',
      message: `Job ${job.id} failed after ${this.maxRetries} attempts`,
      details: { job, error }
    });

    // Notify user in dashboard
    await notifications.create({
      user_id: job.user_id,
      type: 'docling_failed',
      title: 'Document processing failed',
      message: `Failed to process: ${path.basename(job.file_path)}`,
      action_url: `/libraries/${job.library_id}`
    });
  }
}
```

2. Orchestrator n8n Workflow Failure Handling
```typescript
// services/orchestrator/src/n8n-adapter.ts
export class N8nAdapter {
  private timeout = 300000; // 5 minutes

  async triggerWorkflow(
    workflowName: string,
    data: any,
    options?: { timeout?: number; retries?: number }
  ): Promise<any> {
    const maxRetries = options?.retries || 2;
    const timeout = options?.timeout || this.timeout;

    for (let attempt = 1; attempt <= maxRetries; attempt++) {
      try {
        const result = await Promise.race([
          this.executeWorkflow(workflowName, data),
          this.timeoutPromise(timeout)
        ]);
```

```
      return result;

    } catch (error) {
      if (attempt === maxRetries) {
        // Final failure - log and fallback
        await this.handleWorkflowFailure(workflowName, data, error);
        throw error;
      }

      // Retry with backoff
      await sleep(5000 * attempt);
    }
  }
}

private async handleWorkflowFailure(
  workflowName: string,
  data: any,
  error: Error
) {
  // Log failure details
  await redis.json.set(`workflow:failure:${data.task_id}`, '.', {
    workflow: workflowName,
    task_id: data.task_id,
    error: error.message,
    inputs: data,
    timestamp: new Date().toISOString()
  });

  // Check if we can fallback to direct agent call
  if (this.hasDirectAgentFallback(workflowName)) {
    logger.warn(`n8n workflow ${workflowName} failed, using direct agent`);
    return await this.executeDirectAgent(data);
  }

  throw new Error(`Workflow ${workflowName} failed: ${error.message}`);
}

private timeoutPromise(ms: number): Promise<never> {
  return new Promise((_, reject) => {
    setTimeout(() => reject(new Error('Workflow timeout')), ms);
  });
}
```

```
}
```

3. Embedding Generation Failure Recovery
```typescript
// services/docling/src/embedding.ts
export class EmbeddingGenerator {
  async generateEmbeddings(chunks: DoclingChunk[]): Promise<number[][]> {
    const embeddings: number[][] = [];
    const failedChunks: number[] = [];

    for (let i = 0; i < chunks.length; i++) {
      try {
        const embedding = await this.generateSingleEmbedding(chunks[i].content);
        embeddings.push(embedding);
      } catch (error) {
        logger.error(`Embedding generation failed for chunk ${i}:`, error);
        failedChunks.push(i);

        // Use zero vector as placeholder
        embeddings.push(new Array(768).fill(0));
      }
    }

    // Retry failed chunks with fallback model
    if (failedChunks.length > 0) {
      logger.warn(`Retrying ${failedChunks.length} failed embeddings with fallback`);

      for (const idx of failedChunks) {
        try {
          const embedding = await this.generateWithFallback(chunks[idx].content);
          embeddings[idx] = embedding;
        } catch (error) {
          // Mark chunk as non-searchable
          chunks[idx].searchable = false;
          logger.error(`Chunk ${idx} permanently failed embedding`);
        }
      }
    }

    return embeddings;
  }
}
```

🚦 Concurrency Control & Rate Limiting

1. Task Queue with Priority
```typescript
// services/shared/src/queue.ts
export class TaskQueue {
  private queues: Map<string, PriorityQueue> = new Map();

  async enqueue(
    queueName: string,
    task: Task,
    priority: 'high' | 'normal' | 'low' = 'normal'
  ) {
    const score = this.calculateScore(priority, task);

    await redis.zadd(
      `queue:${queueName}`,
      score,
      JSON.stringify(task)
    );
  }

  async dequeue(queueName: string, count: number = 1): Promise<Task[]> {
    // Pop highest priority items
    const results = await redis.zpopmax(`queue:${queueName}`, count);
    return results.map(r => JSON.parse(r.value));
  }

  private calculateScore(priority: string, task: Task): number {
    const priorityScores = { high: 3, normal: 2, low: 1 };
    const baseScore = priorityScores[priority] * 1000000;
    const timestamp = Date.now();

    return baseScore + timestamp; // FIFO within priority
  }
}
```

2. Per-User Rate Limiting
```typescript
// services/orchestrator/src/rate-limiter.ts
```

```
export class RateLimiter {
  private limits = {
    docling_jobs_per_hour: 50,
    agent_tasks_per_hour: 100,
    cloud_model_calls_per_day: 200
  };

  async checkLimit(userId: string, resource: string): Promise<boolean> {
    const key = `ratelimit:${userId}:${resource}`;
    const limit = this.limits[resource];
    const window = this.getWindow(resource);

    // Sliding window counter
    const count = await redis.incr(key);

    if (count === 1) {
      await redis.expire(key, window);
    }

    if (count > limit) {
      // Rate limit exceeded
      await this.notifyUserRateLimit(userId, resource);
      return false;
    }

    return true;
  }

  async getUsage(userId: string, resource: string): Promise<Usage> {
    const key = `ratelimit:${userId}:${resource}`;
    const count = await redis.get(key) || 0;
    const ttl = await redis.ttl(key);
    const limit = this.limits[resource];

    return {
      used: count,
      limit: limit,
      remaining: limit - count,
      resets_in: ttl
    };
  }
}
```

3. Library-Level Concurrency Control

```typescript
// services/docling/src/library-lock.ts
export class LibraryLock {
  private maxConcurrentPerLibrary = 3;

  async acquireLock(libraryId: string): Promise<boolean> {
    const key = `lock:library:${libraryId}`;
    const current = await redis.get(key) || 0;

    if (current >= this.maxConcurrentPerLibrary) {
      return false; // Library at capacity
    }

    await redis.incr(key);
    await redis.expire(key, 3600); // 1 hour max

    return true;
  }

  async releaseLock(libraryId: string) {
    const key = `lock:library:${libraryId}`;
    await redis.decr(key);
  }
}
```

4. Ollama Server Rate Limiting

```typescript
// services/shared/src/ollama/client.ts
export class OllamaClient {
  private queue = new PQueue({ concurrency: 4 }); // Max 4 concurrent requests

  async generate(request: GenerateRequest): Promise<GenerateResponse> {
    return await this.queue.add(async () => {
      // Check if local server is available
      if (!(await this.isHealthy())) {
        throw new Error('Ollama server unhealthy');
      }

      // Execute request
      return await this.executeGenerate(request);
    });
  }
```

```
  private async isHealthy(): Promise<boolean> {
    try {
      const response = await fetch(`${this.baseUrl}/api/tags`, {
        timeout: 5000
      });
      return response.ok;
    } catch {
      return false;
    }
  }
}
```

📊 Model Auto-Selection with Audit Trail

1. Model Selection Decision Tree
```typescript
// services/orchestrator/src/model-selector.ts
export class ModelSelector {
  async selectModel(
    agent: AgentType,
    task: Task,
    context: ModelSelectionContext
  ): Promise<ModelDecision> {
    const decision: ModelDecision = {
      selected_model: '',
      reasoning: [],
      cost_estimate: 0,
      fallback_chain: []
    };

    // Step 1: Default to local
    decision.selected_model = 'gpt-oss:20b';
    decision.reasoning.push('Starting with local quantized model');

    // Step 2: Check complexity indicators
    const complexity = await this.assessComplexity(task);

    if (complexity.score > 7) {
      decision.selected_model = this.getCloudModel(agent);
      decision.reasoning.push(`High complexity (${complexity.score}/10) requires cloud model`);
```

```
    decision.cost_estimate = this.estimateCost(decision.selected_model, task);
  }

  // Step 3: Check previous failures
  const history = await this.getTaskHistory(task.project_id, agent);
  if (history.recent_failures > 2) {
    decision.selected_model = this.getCloudModel(agent);
    decision.reasoning.push(`${history.recent_failures} recent failures, upgrading model`);
  }

  // Step 4: User preferences
  if (context.user_preferences?.prefer_cloud) {
    decision.selected_model = this.getCloudModel(agent);
    decision.reasoning.push('User preference: cloud models');
  }

  // Step 5: Budget check
  if (decision.selected_model.includes('cloud')) {
    const budget = await this.checkBudget(context.user_id);
    if (!budget.can_afford) {
      decision.selected_model = 'gpt-oss:20b';
      decision.reasoning.push('Budget limit reached, using local model');
    }
  }

  // Store decision for audit
  await this.recordDecision(task.id, decision);

  return decision;
}

private async assessComplexity(task: Task): Promise<ComplexityScore> {
  const indicators = {
    query_length: task.query.length,
    has_code: /```/.test(task.query),
    requires_reasoning: /why|how|explain|analyze/i.test(task.query),
    multi_step: task.execution_plan?.steps.length > 3,
    dependencies: task.execution_plan?.dependencies.length || 0
  };

  let score = 0;
  const reasons = [];

  if (indicators.query_length > 500) {
```

```
    score += 2;
    reasons.push('Long query requires more context');
  }

  if (indicators.has_code) {
    score += 3;
    reasons.push('Code generation requires specialized model');
  }

  if (indicators.requires_reasoning) {
    score += 2;
    reasons.push('Reasoning task benefits from larger model');
  }

  if (indicators.multi_step) {
    score += 2;
    reasons.push('Multi-step task needs better planning');
  }

  return { score, reasons, indicators };
 }

 private async recordDecision(taskId: string, decision: ModelDecision) {
  await redis.json.set(`model_decision:${taskId}`, '', {
    task_id: taskId,
    decision: decision,
    timestamp: new Date().toISOString()
  });

  // Aggregate stats for optimization
  await redis.hincrby('model_stats', decision.selected_model, 1);
 }
}
```

2. Cost Tracking & Budget Alerts

```typescript
// services/orchestrator/src/cost-tracker.ts
export class CostTracker {
 private costs = {
  'gpt-oss:20b': 0, // Local is free
  'gpt-oss:120b-cloud': 0.002, // $ per 1k tokens
  'devstral-2:123b-cloud': 0.003,
  'deepseek-v3.1:671b-cloud': 0.004,
```

```
    // ... other models
  };

  async trackUsage(
    userId: string,
    model: string,
    tokensUsed: number
  ) {
    const cost = (tokensUsed / 1000) * (this.costs[model] || 0);

    // Increment user's spend
    await redis.hincrbyfloat(`cost:${userId}:daily`, model, cost);
    await redis.hincrbyfloat(`cost:${userId}:monthly`, model, cost);

    // Check budget thresholds
    const dailySpend = await this.getDailySpend(userId);
    const monthlySpend = await this.getMonthlySpend(userId);

    // Alert at 80% of budget
    if (dailySpend > 8.00) { // $8 of $10 daily budget
      await this.sendBudgetAlert(userId, 'daily', dailySpend);
    }

    if (monthlySpend > 80.00) { // $80 of $100 monthly
      await this.sendBudgetAlert(userId, 'monthly', monthlySpend);
    }
  }

  async getUsageReport(userId: string, period: 'daily' | 'monthly'): Promise<CostReport> {
    const key = `cost:${userId}:${period}`;
    const breakdown = await redis.hgetall(key);

    const total = Object.values(breakdown).reduce((sum, cost) => sum + parseFloat(cost), 0);

    return {
      period,
      total_cost: total,
      breakdown: breakdown,
      generated_at: new Date().toISOString()
    };
  }
}
```

🎯 Quality Considerations for Quantized Models

1. Multi-GPU Sharded Inference Config

```yaml
# services/orchestrator/config/models.yml
local_inference:
  gpt-oss:20b:
    quantization: Q4_K_M
    gpu_layers: 40  # Offload 40 layers to GPU
    context_size: 8192
    memory_required: 14GB

  gpt-oss:20b-fp8:
    quantization: FP8
    gpu_layers: -1  # All layers on GPU
    gpu_split: [0.6, 0.4]  # 60% GPU0, 40% GPU1
    context_size: 16384
    memory_required: 22GB  # Fits in 16GB + 32GB setup
    note: "Requires dual-GPU setup with tensor parallelism"

model_quality_tiers:
  tier1_fast:
    model: gpt-oss:20b
    quantization: Q4_K_M
    use_case: "Quick responses, simple tasks"
    expected_quality: 6/10

  tier2_balanced:
    model: gpt-oss:20b
    quantization: Q5_K_M
    use_case: "Standard tasks, good quality"
    expected_quality: 7/10

  tier3_quality:
    model: gpt-oss:20b-fp8
    quantization: FP8
    use_case: "Complex reasoning, code generation"
    expected_quality: 8/10
    gpu_required: 2

  tier4_cloud:
    model: gpt-oss:120b-cloud
```

```
    use_case: "Highest quality, production code"
    expected_quality: 9/10
```

2. Quality Monitoring & Auto-Upgrade
```typescript
// services/orchestrator/src/quality-monitor.ts
export class QualityMonitor {
  async evaluateResponse(
    taskId: string,
    response: AgentResponse,
    model: string
  ): Promise<QualityScore> {
    const score = {
      completeness: 0,
      coherence: 0,
      correctness: 0,
      overall: 0,
      should_upgrade: false
    };

    // Check response quality indicators
    score.completeness = this.checkCompleteness(response);
    score.coherence = await this.checkCoherence(response);
    score.correctness = await this.checkCorrectness(response);

    score.overall = (score.completeness + score.coherence + score.correctness) / 3;

    // Auto-upgrade decision
    if (score.overall < 6.0 && !model.includes('cloud')) {
      score.should_upgrade = true;

      await this.recordQualityIssue(taskId, {
        model: model,
        score: score,
        recommendation: 'Upgrade to cloud model'
      });
    }

    return score;
  }

  private checkCompleteness(response: AgentResponse): number {
    // Check if response addresses the query
```

```
    const hasResult = response.result && Object.keys(response.result).length > 0;
    const hasExplanation = response.explanation && response.explanation.length > 100;
    const hasSources = response.sources && response.sources.length > 0;

    let score = 0;
    if (hasResult) score += 4;
    if (hasExplanation) score += 3;
    if (hasSources) score += 3;

    return score;
  }
}
```

🎨 Enhanced User-Facing UX

1. Libraries Tab - Full Implementation
```typescript
// services/dashboard/src/app/libraries/page.tsx
export default function LibrariesPage() {
  const [libraries, setLibraries] = useState<Library[]>([]);
  const [uploading, setUploading] = useState(false);

  return (
    <div className="p-6">
      <div className="flex justify-between items-center mb-6">
        <h1 className="text-2xl font-bold">Document Libraries</h1>
        <button onClick={() => setShowAddModal(true)}>
          + Add Library
        </button>
      </div>

      <div className="grid grid-cols-3 gap-4">
        {libraries.map(lib => (
          <LibraryCard
            key={lib.id}
            library={lib}
            onUpload={(files) => handleUpload(lib.id, files)}
            onRescan={() => triggerRescan(lib.id)}
            onDelete={() => deleteLibrary(lib.id)}
          />
        ))}
```

```
      </div>

      {/* File Upload Modal */}
      <UploadModal
        isOpen={uploadModalOpen}
        onUpload={handleFileUpload}
        onClose={() => setUploadModalOpen(false)}
      />
    </div>
  );
}

function LibraryCard({ library, onUpload, onRescan, onDelete }) {
  return (
    <div className="border rounded-lg p-4">
      <div className="flex items-center justify-between mb-3">
        <div className="flex items-center gap-2">
          <FolderIcon />
          <h3 className="font-semibold">{library.name}</h3>
        </div>
        <StatusBadge status={library.scan_status} />
      </div>

      <div className="space-y-2 text-sm text-slate-600">
        <div className="flex justify-between">
          <span>Documents:</span>
          <span className="font-medium">{library.document_count}</span>
        </div>
        <div className="flex justify-between">
          <span>Last scanned:</span>
          <span>{formatRelative(library.last_scanned)}</span>
        </div>
        <div className="flex justify-between">
          <span>Size:</span>
          <span>{formatBytes(library.total_size)}</span>
        </div>
      </div>

      {/* Ingestion Progress */}
      {library.scan_status === 'processing' && (
        <ProgressBar
          current={library.processed_count}
          total={library.total_files}
          label="Processing documents"
```

```
        />
      )}

      {/* Actions */}
      <div className="flex gap-2 mt-4">
        <button onClick={() => onUpload([])}>
          <UploadIcon /> Upload
        </button>
        <button onClick={onRescan}>
          <RefreshIcon /> Rescan
        </button>
        <button onClick={() => navigate(`/libraries/${library.id}/search`)}>
          <SearchIcon /> Search
        </button>
        <DropdownMenu>
          <MenuItem onClick={() => navigate(`/libraries/${library.id}/settings`)}>
            Settings
          </MenuItem>
          <MenuItem onClick={onDelete} danger>
            Delete
          </MenuItem>
        </DropdownMenu>
      </div>
    </div>
  );
}
```

2. RAG Query Results with Source Citations
```typescript
// services/dashboard/src/components/RAGResponse.tsx
function RAGResponse({ response }: { response: RAGQueryResult }) {
  return (
    <div className="space-y-4">
      {/* Main Answer */}
      <div className="prose max-w-none">
        <ReactMarkdown>{response.answer}</ReactMarkdown>
      </div>

      {/* Source Citations */}
      <div className="border-t pt-4">
        <h4 className="font-semibold mb-3">Sources</h4>
        <div className="space-y-2">
          {response.sources.map((source, idx) => (
```

```
          <SourceCard key={idx} source={source} />
        ))}
      </div>
    </div>

    {/* Images Referenced */}
    {response.images && response.images.length > 0 && (
      <div className="border-t pt-4">
        <h4 className="font-semibold mb-3">Referenced Images</h4>
        <div className="grid grid-cols-3 gap-4">
          {response.images.map((img, idx) => (
            <ImagePreview key={idx} image={img} />
          ))}
        </div>
      </div>
    )}
  </div>
 );
}

function SourceCard({ source }: { source: Source }) {
 return (
  <div className="bg-slate-50 rounded-lg p-3">
    <div className="flex items-start justify-between">
      <div className="flex-1">
        <div className="flex items-center gap-2 mb-1">
          <DocumentIcon className="text-slate-500" />
          <span className="font-medium text-sm">{source.document_name}</span>
        </div>
        <p className="text-xs text-slate-600 mb-2">
          {source.section_title} • Page {source.page_number}
        </p>
        <p className="text-sm text-slate-700 line-clamp-2">
          {source.excerpt}
        </p>
      </div>
      <button
        onClick={() => openDocument(source.document_id, source.page_number)}
        className="text-blue-600 text-sm"
      >
        View →
      </button>
    </div>
  </div>
```

```
  );
}
```

📚 Documentation & Testing Strategy

1. Comprehensive README Structure
```markdown
# Mother-Harness

## Quick Start
- Prerequisites
- Installation (one-command setup)
- First task walkthrough

## Architecture
- System overview
- Component responsibilities
- Data flow diagrams

## Development
- Local development setup
- Running tests
- Debugging guide

## Deployment
- Homelab deployment guide
- Configuration options
- Scaling strategies

## Agent Development
- Creating new agents
- Agent testing framework
- Best practices

## Contributing
- Code style
- Pull request process
- Issue templates
```

2. Automated Testing Suite

```typescript
// tests/integration/orchestrator.test.ts
describe('Orchestrator Integration', () => {
  test('should create task and execute plan', async () => {
    const query = 'Research vector databases';

    // Create task
    const response = await fetch('http://localhost:8000/api/ask', {
      method: 'POST',
      body: JSON.stringify({ query, user_id: 'test' })
    });

    const { task_id } = await response.json();
    expect(task_id).toBeDefined();

    // Wait for completion
    const result = await waitForTaskCompletion(task_id, 60000);

    expect(result.status).toBe('completed');
    expect(result.result).toBeDefined();
    expect(result.artifacts.length).toBeGreaterThan(0);
  });

  test('should handle n8n workflow failure gracefully', async () => {
    // Simulate n8n down
    await stopN8n();

    const query = 'Generate code';
    const response = await fetch('http://localhost:8000/api/ask', {
      method: 'POST',
      body: JSON.stringify({ query, user_id: 'test' })
    });

    const { task_id } = await response.json();
    const result = await waitForTaskCompletion(task_id, 30000);

    // Should fallback to direct agent
    expect(result.status).toBe('completed');
    expect(result.metadata.used_fallback).toBe(true);
  });
});

// tests/integration/docling.test.ts
describe('Docling Pipeline', () => {
```

```
  test('should ingest PDF and extract images', async () => {
    const testFile = './test-fixtures/sample.pdf';

    // Copy to watch folder
    await fs.copyFile(testFile, '//core4/libraries/test/sample.pdf');

    // Wait for processing
    await sleep(10000);

    // Check Redis for chunks
    const chunks = await redis.ft.search('idx:documents', '@document_name:sample.pdf');

    expect(chunks.total).toBeGreaterThan(0);
    expect(chunks.documents[0].images).toBeDefined();
  });

  test('should retry failed embeddings', async () => {
    // Mock embedding failure
    jest.spyOn(ollama, 'embed').mockRejectedValueOnce(new Error('Timeout'));

    const processor = new DoclingProcessor();
    const result = await processor.generateEmbeddings([
      { content: 'Test chunk 1' },
      { content: 'Test chunk 2' }
    ]);

    // Should have retried and succeeded
    expect(result.length).toBe(2);
    expect(result[0].length).toBe(768);
  });
});
```

3. GitHub Actions CI/CD
```yaml
# .github/workflows/test.yml
name: Test Suite

on: [push, pull_request]

jobs:
  unit-tests:
    runs-on: ubuntu-latest
    steps:
```

```
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '20'
      - run: npm install
      - run: npm test

  integration-tests:
    runs-on: ubuntu-latest
    services:
      redis:
        image: redis/redis-stack:latest
        ports:
          - 6379:6379
    steps:
      - uses: actions/checkout@v3
      - run: npm install
      - run: npm run test:integration

  e2e-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: docker-compose up -d
      - run: npm run test:e2e
      - run: docker-compose down
```

Architecture Version: 1.1 - Production-Ready
Enhanced: Error handling, resiliency, testing, documentation
Status: Ready for Implementation with Enterprise-Grade Quality