Data Standardization Platform using Agentic AI
Developer Guide

B.Tech Final Year Project

by
Lalith Chatala (21JE0508)
Joy (21JE0430)

Department of Computer Science and Engineering
Prof. Chiranjeev Kumar
Indian Institute of Technology (ISM), Dhanbad

May 4, 2025

Contents

Chapter 1

Introduction

DataSync is a comprehensive data structuring framework designed to address key challenges in data management, primarily targeting defense systems but applicable to various domains requiring structured data processing. The system enables users to perform ETL (Extract, Transform, Load) operations through a web-based interface, with features including data transformation, standardization, merging, and conversion between different formats such as CSV, JSON, and XML.

This developer guide serves as the technical reference for developers working on the DataSync codebase. It provides detailed information about the architecture, API endpoints, implementation details, and best practices for extending and maintaining the system. Key features of DataSync include dataset transformation operations such as merge, concatenate, standardize, and split; format conversion operations between CSV and JSON/XML (and vice versa); visual workflow creation through flow diagrams; AI-powered suggestions and an Agentic AI chatbot to assist with data operations; as well as user management and authentication capabilities.

Chapter 2

Architecture Overview

DataSync follows a modern web application architecture with the following components:

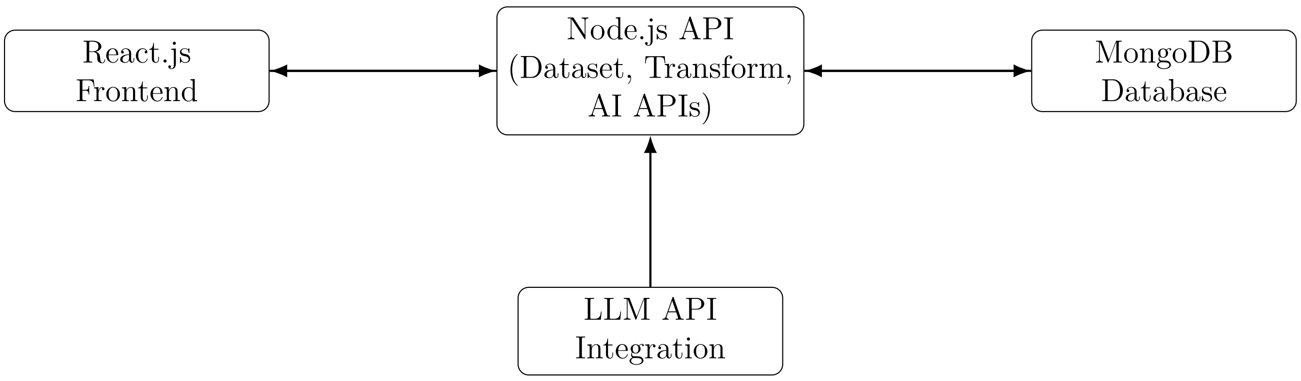
2.1 Frontend

- **Framework**: React.js
- **State Management**: React Hooks and Context API
- **UI Components**: Custom components built for data operations
- **Data Visualization**: React Flow for workflow diagrams
- **Communication**: Axios for REST API calls

2.2 Backend

- **Runtime**: Node.js
- **Database**: MongoDB (NoSQL database for storing datasets, user information, and operation results)
- **Authentication**: JWT (JSON Web Tokens)
- **File Processing**: Custom modules for CSV, JSON, and XML processing
- **AI Integration**: Integration with LLM APIs for suggestions and the Agentic AI chatbot

2.3 System Architecture Diagram



2.4 Data Flow

1. **User Input**: Front-end captures user actions and input parameters.
2. **API Requests**: Front-end sends requests to appropriate backend API endpoints.
3. **Data Processing**: Backend performs requested operations on datasets.
4. **Database Operations**: Results are stored in MongoDB.
5. **Response**: Backend sends operation results or confirmation back to frontend.
6. **Visualization**: Frontend displays results to the user.

Chapter 3

Environment Setup and Installation

3.1 Prerequisites

- Node.js (v14.x or higher)
- MongoDB (v4.4 or higher)
- npm (v6.x or higher) or yarn (v1.22.x or higher)

3.2 Backend Setup

```
# Clone the repository
git clone https://github.com/your-org/datasync.git
cd datasync/backend

# Install dependencies
npm install

# Configure environment variables
cp .env.example .env
# Edit .env with your MongoDB URI and other configuration

# Start the development server
npm run dev
```

3.3 Frontend Setup

```
\# Navigate to frontend directory\\
cd ../frontend

\# Install dependencies\\
npm install

\# Configure API endpoint\\
cp .env.example .env\\
\# Edit .env with your API\_BASE\_URL
\# Start the development server\\
npm start
```

3.4 Environment Variables

Backend `.env` PORT=5000
MONGO_URI=mongodb://localhost:27017/datasync
JWT_SECRET=your_jwt_secret_key
GEMINI_API_KEY=your_gemini_api_key

Chapter 4

Codebase Structure

4.1 Backend structure

```
server/
  package.json
  server.js
  .gitignore
  controllers/
    aiController.js
    authController.js
    fileController.js
  helpers/
    fileHelper.js
  models/
    Chat.js
    File.js
    User.js
  routes/
    aiRoutes.js
    authRoutes.js
    fileRoutes.js
  services/
    aiFileService.js
    aiService.js
    fileService.js
    generateFileId.xml
    prompt.xml
    prompt2.xml
    suggestionPrompt.xml
    suggestionsService.js
    titlePrompt.xml
```

4.2 Frontend Structure

```
src/
|- App.js, App.css, index.js, index.css, setupTests.js, reportWebVitals.js
|- components/
|  |- Features/
|  |  |- AI/
|  |  |  |- AI.jsx, AI.module.css
|  |  |  |- components/ → ChatMessageList, PromptInput
|  |  |  |- hooks/ → useAIChat.js
|  |  |  \-- RightSidebar/ → RightSidebar.jsx, .module.css
|  |  \-- [Concatenate | Convert | ConvertBack | Merge | Split | Standardize]/
|  |     \-- Each: .jsx, .module.css
|  |- Layout/
|  |  |- Layout.jsx, .module.css
|  |  \-- Navbar/, Sidebar/ → *.jsx, .module.css
|  |- Pages/
|  |  |- Dashboard/ → Dashboard.jsx, .module.css
|  |  |- Datasets/, Results/ → DatasetModal, DatasetTable, SkeletonLoader, Suggestions
|  |  |- FlowDiagrams/
|  |  |  |- FlowDiagrams.jsx, .module.css
|  |  |  |- hooks/ → apiOperations, useFlowLogic, useFlowUI, validations
|  |  |  |- Modal/ → Modal, PeekModal, SaveTemplateModal, ViewTemplatesModal
|  |  |  |- Nodes/ → ActionNode, DatasetNode, OutputNode, Tooltip
|  |  |  |- RightSideBar/ → ActionCard, ActionsTab, DataTab, OutputTab, ParametersVie
|  |  |  |- SidebarActions/ → SidebarConcatenate, SidebarMerge, SidebarSplit, SidebarS
|  |  |  \-- utils/ → resizeObserverUtils.js, utils.js
|  |  \-- Login/, Register/ → *.jsx, .module.css
|  \-- UI/
|     |- DataTable/ → DataTable, DataTableSkeleton, useSearchQuery
|     |- Dropdown/ → Dropdown.jsx, .module.css
|     \-- UploadModal/ → UploadModal.jsx, .module.css
|- context/ → UserContext.jsx
\-- hooks/ → usePagination.jsx
```

Chapter 5

FlowDiagram Overview

5.1 File Overview

The codebase comprises three main files: `FlowDiagrams.jsx`, which is the main component where the flow diagram is rendered using **React Flow**; `RightSideBar.jsx`, which enables users to add nodes, configure ETL operations, and interact with the flowchart dynamically; and `useFlowLogic.jsx`, a custom React hook that encapsulates all the logic related to managing nodes, edges, their data, and flow state.

5.2 Architecture and Workflow

The system integrates tightly with React Flow. The `FlowDiagrams` component sets up the core diagram interface using `ReactFlow` for rendering, `ReactFlowProvider` for context, and the hooks `useReactFlow`, `useNodesState`, and `useEdgesState` for state management. Nodes and edges are customized via types and provided by the `useFlowLogic` hook. Supported node types include dataset nodes (regular nodes with a blue border), temporary nodes (created via `RightSideBar` with a red border), and action nodes (Concatenate, Merge, Split, Standardize).

5.3 Core Components

The `FlowDiagrams` component renders the main diagram, mini-map, background grid, and control panel. It uses the `useFlowLogic` hook to handle drag-and-drop, manage nodes and edges, and validate connections. Key features include the `onConnect` and `onEdgesChange` handlers to maintain edge integrity, support for node-click events to trigger parameter UIs, and built-in background grid and zoom/pan controls.

The `RightSideBar` component provides the UI for adding new nodes and selecting ETL actions: an input field for naming a temporary node, a dropdown for choosing an action node type, and, once edges are connected, a dynamic parameter panel rendering the appropriate component (`Concatenate`, `Merge`, `Split`, `Standardize`, `Convert`, or `ConvertBack`).

The `useFlowLogic` hook encapsulates all the orchestration logic—adding nodes, managing node/edge state, validating edges, and wiring up action nodes. Its API includes functions like `addTemporaryNode(name)`, `addActionNode(type)`, event handlers `onConnect`, `onEdgeUpdate`, `onEdgesChange`, and `validateEdge(edge)`, which ensures correct dataset→action direction. When an action node is clicked, it finds the connected dataset nodes and renders the corresponding parameter UI, binding data via edge references.

5.4 Usage Flow

1. **Start Flow Diagram**: loads a blank canvas with all controls.
2. **Add Nodes via Sidebar**: add temporary dataset nodes by name and action nodes via the dropdown.
3. **Connect Nodes**: drag edges between dataset and action nodes; invalid connections are rejected.
4. **Configure Action**: click an action node to display and set parameters based on connected datasets.
5. **Visual Aids**: use the minimap, background grid, and zoom/pan controls for navigation.

Chapter 6

Backend Algorithm

Backend Algorithm
The backend handles data manipulation and format conversion using various algorithms. Below is an overview of the main functions.
Concatenation: First we extract the necessary data (dataset, columns, delimiter) from the request and validate them. We then locate the specified file and read its CSV content, parsing it into rows. For each row, the values of the specified columns are concatenated with the delimiter, added as a new column, and the original columns removed. Finally, the modified rows are converted back to CSV format, saved as a new file, and a success message returned. This enables users to create a new CSV with concatenated columns based on input specifications.

```
function concatenateColumns(dataset, columns, finalName, delimiter):
    if not dataset or not columns or not finalName or not delimiter:
        return error("Required fields missing")
    file = findFileByName(dataset)
    if not file:
        return error("File not found")
    rows = parseCSV(decodeBase64(file.data))
    for row in rows:
        value = join([row[col] for col in columns], delimiter)
        row[finalName] = value
        for col in columns:
            delete row[col]
    saveNewFile(convertToCSV(rows), "concat_result.csv")
    return success("Concatenation complete")
```

Merge: We start by extracting dataset names and join columns from the request body, validating inputs and locating each dataset file. Both files are decoded and parsed into rows. A full outer join is performed on the two datasets based on the specified columns. The merged data is converted back to CSV, saved as a new file, and a success message returned. This function allows users to merge two datasets on shared columns for seamless data integration.

```
function mergeDatasets(dataset1, dataset2, col1, col2):
    if not dataset1 or not dataset2 or not col1 or not col2:
        return error("Required fields missing")
    file1 = findFileByName(dataset1)
    file2 = findFileByName(dataset2)
    if not file1 or not file2:
        return error("Datasets not found")
    rows1 = parseCSV(decodeBase64(file1.data))
    rows2 = parseCSV(decodeBase64(file2.data))
    result = fullOuterJoin(rows1, rows2, col1, col2)
    if not result:
        return error("No matching rows")
    saveNewFile(convertToCSV(result), "merge_result.csv")
    return success("Merge complete")
```

Standardize Column: This function standardizes values in a specific column according to provided mappings. After validating inputs and locating the dataset file, it decodes and parses the CSV. Each row's target column is compared against the 'before' values in the mappings; matches are replaced with the corresponding 'after' values. The updated rows are converted back to CSV, saved as a new file, and a success message returned.

```
function standardizeColumn(dataset, column, mappings, outputName):
    if not dataset or not column or not mappings or not outputName:
        return error("Required fields missing or invalid mappings")
    file = findFileByName(dataset)
    if not file:
        return error("Dataset not found")
    rows = parseCSV(decodeBase64(file.data))
    for row in rows:
        for map in mappings:
            if row[column] in map.before:
                row[column] = map.after
    saveNewFile(convertToCSV(rows), outputName)
    return success("Standardization complete")
```

General Column Split: We locate the file by ID, decode and parse it into rows, then apply split operations on designated columns. Each column's value is split by the given delimiter up to a specified count, with parts assigned to new columns. The modified rows are converted back to CSV and saved.

```
function splitColumnsGeneral(fileId, splits, outputName):
    file = findFileById(fileId)
    rows = parseCSV(decodeBase64(file.data))
    for row in rows:
        for split in splits:
            parts = row[split.column].split(split.delimiter, split.count + 1)
            for i, name in enumerate(split.columnNames):
                row[name] = parts[i]
    saveNewFile(convertToCSV(rows), outputName)
    return success("General split complete")
```

Address Split: After locating and parsing the file, each row's address field is split by commas. A regex extracts the pincode, which is used to fetch district, state, and country data via an external API. The split parts and enriched data populate new columns before the rows are converted back to CSV and saved.

```
function splitAddress(fileId, addressField, outputName):
    file = findFileById(fileId)
    rows = parseCSV(decodeBase64(file.data))
    for row in rows:
        parts = row[addressField].split(",")
        pin = extractPincode(parts[-1])
        loc = fetchLocation(pin)
        row["Street"] = join(parts[:-1], ",")
        row["PostOffice"] = loc.postOffice
        row["District"] = loc.district
        row["State"] = loc.state
        row["Country"] = loc.country
    saveNewFile(convertToCSV(rows), outputName)
    return success("Address split complete")
```

Convert to CSV: The function checks if input is JSON or XML. For JSON, it parses and maps keys to CSV rows. For XML, it converts XML to JSON, flattens nested structures, then maps to CSV. The resulting CSV buffer is saved.

```
function convertToCSV(type, data):
    if type == "json":
        obj = parseJSON(data)
        headers = extractKeys(obj[0])
        rows = [mapValues(item, headers) for item in obj]
    else if type == "xml":
        obj = flatten(parseXML(data))
        headers = extractKeys(obj[0])
        rows = [mapValues(item, headers) for item in obj]
    else:
        return error("Unsupported type")
    saveNewFile(buildCSV(headers, rows), "converted.csv")
    return success("CSV conversion complete")
```

Convert Back (CSV to JSON/XML): After parsing CSV into JSON objects, the function returns either a JSON string or constructs an XML document with each row as a nested element under a root tag, based on the desired output format.

```
function convertBack(fileData, format):
    jsonObj = parseCSVToJson(fileData)
    if format == "json":
        return stringify(jsonObj)
    else if format == "xml":
        xml = wrapInRoot(toXmlElements(jsonObj))
        return xml
    else:
        return error("Unsupported format")
```

Chapter 7

API Documentation

7.1 File API

The File API base path is `/api/file`. It provides endpoints for uploading, retrieving, deleting, and transforming files.

Upload File (POST `/api/file/upload`): Uploads a CSV, JSON, or XML file to the database. The request body must include the original filename, MIME type, file buffer (base64-encoded), optional description, and user ID.

```
# Request
{
  "originalname": "file.csv",
  "mimetype": "text/csv",
  "buffer": "<file-buffer>",
  "description":"Optional file description",
  "userId": "USER_ID"
}

# Response
{
  "_id": "file_id",
  "originalName":"file.csv",
  "contentType": "text/csv",
  ...
}
```

Get Dataset by ID (GET `/api/file/dataset/:datasetId`): Retrieves a dataset file using its unique ID; returns the file metadata and data buffer.

Get All Datasets (User Files) (GET `/api/file/datasets/:userId`): Fetches all original (non-result) datasets uploaded by the specified user.

Get Result Datasets (GET `/api/file/results/:userId`): Fetches all result datasets generated by ETL actions for the specified user.

Get All Datasets (Input + Result) (GET `/api/file/all/:userId`): Retrieves both original and result datasets for the specified user.

Delete Dataset (DELETE `/api/file/:datasetId`): Permanently deletes the dataset identified by `datasetId`.

Concatenate Columns (POST `/api/file/concatenate`): Merges multiple columns into one using a delimiter. Request body:

```
{
  "dataset": "file_id",
  "columns": ["firstName","lastName"],
  "finalColumnName": "fullName",
  "delimiter": " ",
  "outputFileName": "concatenated.csv",
  "description": "Merged full names"
}
```

Merge Datasets (Full Outer Join) (POST `/api/file/merge`): Performs a full outer join on two datasets. Request body:

```
{
  "dataset1": "file1_id",
  "dataset2": "file2_id",
  "column1": "id",
  "column2": "id",
  "outputFileName":"merged.csv",
  "description": "Merged on ID"
}
```

Standardize Column (POST `/api/file/standardize`): Replaces values in a column based on mapping arrays. Request body:

```
{
  "datasetId": "file_id",
  "column": "state",
  "mappings": [
    { "before":["CA","Calif."], "after":"California" }
  ],
  "outputFileName": "standardized.csv",
  "description": "Cleaned state names"
}
```

Convert JSON/XML to CSV (POST `/api/file/convert`): Converts JSON or XML data into a CSV file. Request body:

```
{
  "fileType": "json" | "xml",
  "fileData": "{...}",
  "userId": "USER_ID",
  "originalFileName":"converted"
}
```

Convert CSV to JSON or XML (POST `/api/file/convert-back`): Converts CSV data back to JSON or XML. Request body:

```
{
  "convertTo": "json" | "xml",
  "fileData": "<csv content as string>"
}
```

Split Columns (POST `/api/file/split`): Splits a column into multiple new columns by delimiter. Request body:

```
{
  "fileId": "file_id",
  "splits": [
    {
      "col": "fullName",
      "delimiter": " ",
      "numDelimiters": 1,
      "columnNames": ["firstName","lastName"]
    }
  ],
  "outputFileName":"split.csv",
  "description": "Split names"
}
```

Split Address & Enrich (with Pincode) (POST `/api/file/split-address`): Splits an address field and adds location info via an external pincode API. Request body:

```
{
  "fileId": "file_id",
  "addressName": "address",
  "outputFileName": "enriched.csv",
  "description": "Address cleaned and enriched"
}
```

Rename Dataset (PUT `/api/file/rename/:datasetId`): Updates the original filename. Request body:

```
{
  "newName":"updated_name.csv"
}
```

7.2 AI Agent API

The AI Agent API base path is `/api/ai`. It exposes endpoints for conversational and suggestion services.
AI Chat (POST `/api/ai/chat`): Parses the user prompt and chat history, generates XML-based system instructions including dataset metadata and format constraints, maintains a stateful session with the AI model, and loops until a final answer or action is detected.

```
function runChainOfThought(prompt, history):
    instr = buildXMLInstructions(prompt, history)
    session = startAIModel(instr)
    while not done:
        response = session.next()
        if response.containsAction():
            handleAction(response)
        else if response.containsAnswer():
            return response.answer
        history.append(response)
```

Handle Action (internal): Maps action names to service functions and executes with parameters extracted from XML.

```
const ACTION_MAP = {
  "merge": mergeDatasetsService,
  "concatenate": concatenateColumnsService,
  "split": splitColsService,
  "standardize": standardizeColumnService
};

function handleAction(actionXML):
    name, params = parseAction(actionXML)
    service = ACTION_MAP[name]
    result = service(params)
    return result
```

Chapter 8

AI Agents Services and Models

8.1 Routes

The `routes` directory contains three files. `aiRoutes.js` exposes endpoints for AI-related operations such as generating responses and thought suggestions, relying on `aiService.js` and XML prompt templates; `authRoutes.js` handles user authentication (registration and login) using JWT or session-based strategies; and `fileRoutes.js` manages file upload, retrieval, and conversion operations via `fileService.js`.

```
# aiRoutes.js
POST /api/ai/generate -> generateResponseService(prompt, history)
POST /api/ai/suggestions -> getSuggestionsService(conversationId)

# authRoutes.js
POST /api/auth/register -> registerUserService(userData)
POST /api/auth/login -> loginUserService(credentials)

# fileRoutes.js
POST /api/file/upload -> uploadFileService(fileData)
GET /api/file/:id -> getFileService(fileId)
```

8.2 Services

The `services` layer encapsulates reusable business logic. `aiFileService.js` reads and parses XML prompt templates; `aiService.js` interfaces with the AI model (e.g., OpenAI GPT), loading prompts, injecting dynamic context, and returning responses; `fileService.js` handles file storage operations such as uploading buffers to MongoDB and retrieving metadata; and `suggestionsService.js` generates suggestion prompts based on previous messages.

```
# aiFileService.js
function loadPrompt(templateName):
    xml = readFile(‘/services/${templateName}.xml‘)
    return parseXML(xml)

# aiService.js
function callAIModel(promptXML, context):
    response = openAI.complete(promptXML, context)
    return response.text

# fileService.js
function uploadFile(buffer, metadata):
    saveToDatabase(buffer, metadata)
    return fileRecord

# suggestionsService.js
function getSuggestions(chatHistory):
    prompt = loadPrompt('suggestionPrompt')
    return callAIModel(prompt, chatHistory)
```

8.3 Models

The `models` directory defines Mongoose schemas. `Chat.js` models user chat sessions with fields for messages (text, sender, ETL context, thought flags, approval status) and timestamps; `File.js` models uploaded files storing original name, binary data, MIME type, optional description, result flag, and associated user ID.

```
# Chat.js
schema Chat {
    user: ObjectId
    title: String
    messages: [
      { text: String, sender: 'user'|'ai', datasetData: Mixed,
        isThought: Boolean, isFinal: Boolean,
        approved: Boolean, rejected: Boolean }
    ]
    createdAt: Date
}

# File.js
schema File {
    originalName: String
    data: Buffer
    contentType: String
    description: String
    result: Boolean
    userId: ObjectId
}
```

8.4 XML Prompt Files

The `/services` directory contains XML templates defining AI instructions: `prompt.xml` and `prompt2.xml` for base prompts; `generateFileId.xml` for file context queries; `suggestionPrompt.xr` for thought generation; and `titlePrompt.xml` for chat title creation. These templates are dynamically populated and passed to the AI service for processing.

Chapter 9

Extending and Customizing DataSync

Extending and Customizing DataSync
DataSync is designed with extensibility in mind. To add a new transformation operation, start by creating a service module in the backend that implements the core logic. For example, in `services/newTransformation.service.js` you would write:

```
async function newTransformation(dataset, parameters):
    // Read and parse the dataset
    rows = parseCSV(decodeBase64(findFile(dataset).data))
    // Apply transformation to each row
    transformedRows = []
    for row in rows:
        transformedRow = applyTransformationLogic(row, parameters)
        transformedRows.append(transformedRow)
    // Save and return the result
    return saveNewFile(convertToCSV(transformedRows), parameters.outputFileName
        )
```

Next, expose this logic via an API endpoint. In `routes/transform.routes.js`, define:

```
POST /api/transform/new-transformation:
    try:
        result = newTransformationService(req.body.dataset, req.body.parameters
            )
        return 200, result
    except error:
        return 500, { error: error.message }
```

On the frontend, add a component to collect user input and call the new endpoint. In `src/components/transformations/NewTransformation.jsx`:

```
function NewTransformation():
    dataset = useState("")
    parameters = useState({})
    outputName = useState("")
    onSubmit():
        result = transformationService.performNewTransformation(dataset,
            parameters, outputName)
        // Handle success or error

    render:
        form(onSubmit)
            input(name="dataset")
            // fields for parameters
            input(name="outputName")
            button("Transform")
```

Finally, integrate the new operation into the Flow Diagram by updating the action registry and node renderer. In `src/components/flow/actionTypes.js`:

```
ACTION_TYPES["NEW_TRANSFORMATION"] = "new-transformation"
```

and in `ActionNode.jsx` add support for the `new-transformation` type.

To enhance AI capabilities, extend the NLP intent definitions in `ai/training/intents.js`, implement the handler in `ai/handlers/newTransformationHandler.js`, and register it in `ai/agent.js`. For instance:

```
# intents.js
intents.append({
    name: "new_transformation",
    examples: ["perform new transformation on {dataset}", "apply new operation
        "],
    parameters: ["dataset", "param1", "param2"]
})

# newTransformationHandler.js
function handleNewTransformation(params):
    result = transformService.newTransformation(params.dataset, params)
    return { type: "transformation_result", data: result, message: "Completed"
        }

# agent.js
intentHandlers["new_transformation"] = handleNewTransformation
```

To create custom flow actions, define the action in `actionsRegistry.js`, build a React configuration component, and map it in `ActionConfigFactory.jsx`:

```
# actionsRegistry.js
actions["customAction"] = {
    label: "Custom Action",
    parameters: [{ name: "param1", type: "string" }, { name: "param2", type: "
        number" }],
    execute: async (params) => performCustomLogic(params)
}

# CustomActionConfig.jsx
function CustomActionConfig({ node, onUpdate }):
    params = useState(node.data.parameters || {})
    onChange(name, value): onUpdate({ ...params, [name]: value })
    render form fields for each parameter

# ActionConfigFactory.jsx
ACTION_COMPONENTS["customAction"] = CustomActionConfig
```

For debugging, enable detailed logs in backend services and use a logger utility in the frontend. For example, in a backend service:

```
DEBUG = env.DEBUG || false
function debugLog(msg, data):
    if DEBUG: console.log("[DEBUG]", msg, data)

async function mergeDatasets(req):
    debugLog("Merge request", req.body)
    // logic...
    debugLog("Merge result", result)
    return result
```

and in the frontend:

```
# logger.js
logLevels = { ERROR:0, WARN:1, INFO:2, DEBUG:3 }
currentLevel = env.LOG_LEVEL || logLevels.INFO

function debug(msg, data):
    if currentLevel >= logLevels.DEBUG:
        console.debug("[DEBUG]", msg, data)
```

Chapter 10

Contribution Guidelines

Setting Up the Development Environment. Clone the repository, install dependencies for both server and client, set up environment variables, and start the development servers.

```
# Clone the repository
git clone https://github.com/your-organization/DataMap_ETL_Tool
cd DataMap_ETL_Tool

# Install dependencies
cd server
npm install # Server dependencies
cd ../client
npm install # Client dependencies

# Set up environment variables
cp .env.example .env
# Edit .env with your settings

# Start development servers
# In one terminal:
cd server
nodemon server.js
# In another terminal:
cd client
npm run start
```

Code Contribution Process. Create feature branches, commit your changes, run tests, push to origin, and open a pull request following the project's PR template.

```
# Code Contribution Process
git checkout -b feature/your-feature-name

# Make changes and commit
git add .
git commit -m "Description of your changes"

# Run tests
npm test
```

```
# Push your branch
git push origin feature/your-feature-name

# Create Pull Request:
# - Go to the repository on GitHub
# - Click "New pull request"
# - Select your branch
# - Fill in the PR template
```

Pull Request Guidelines. Ensure each PR contains a clear description of changes, screenshots for UI updates, new or updated tests, necessary documentation updates, and passes all CI checks.

1. A clear description of changes
2. Screenshots for UI changes
3. New or updated tests
4. Documentation updates if needed
5. Passing CI checks

Chapter 11

Resources and Support

Learning Resources. A curated list of official documentation and tutorials to help you get up to speed with the technologies used in DataSync.

Documentation Links

- MongoDB Documentation: https://docs.mongodb.com/
- Express.js Guide: https://expressjs.com/en/guide/routing.html
- React Documentation: https://reactjs.org/docs/getting-started.html
- React Flow Tutorial: https://reactflow.dev/
- **ReAct: Synergizing Reasoning and Acting in Language Models** (research paper): https://arxiv.org/abs/2210.03629

Community & Support

- Lalith Chatala: clalith257@gmail.com
- Joy: j26122003@gmail.com