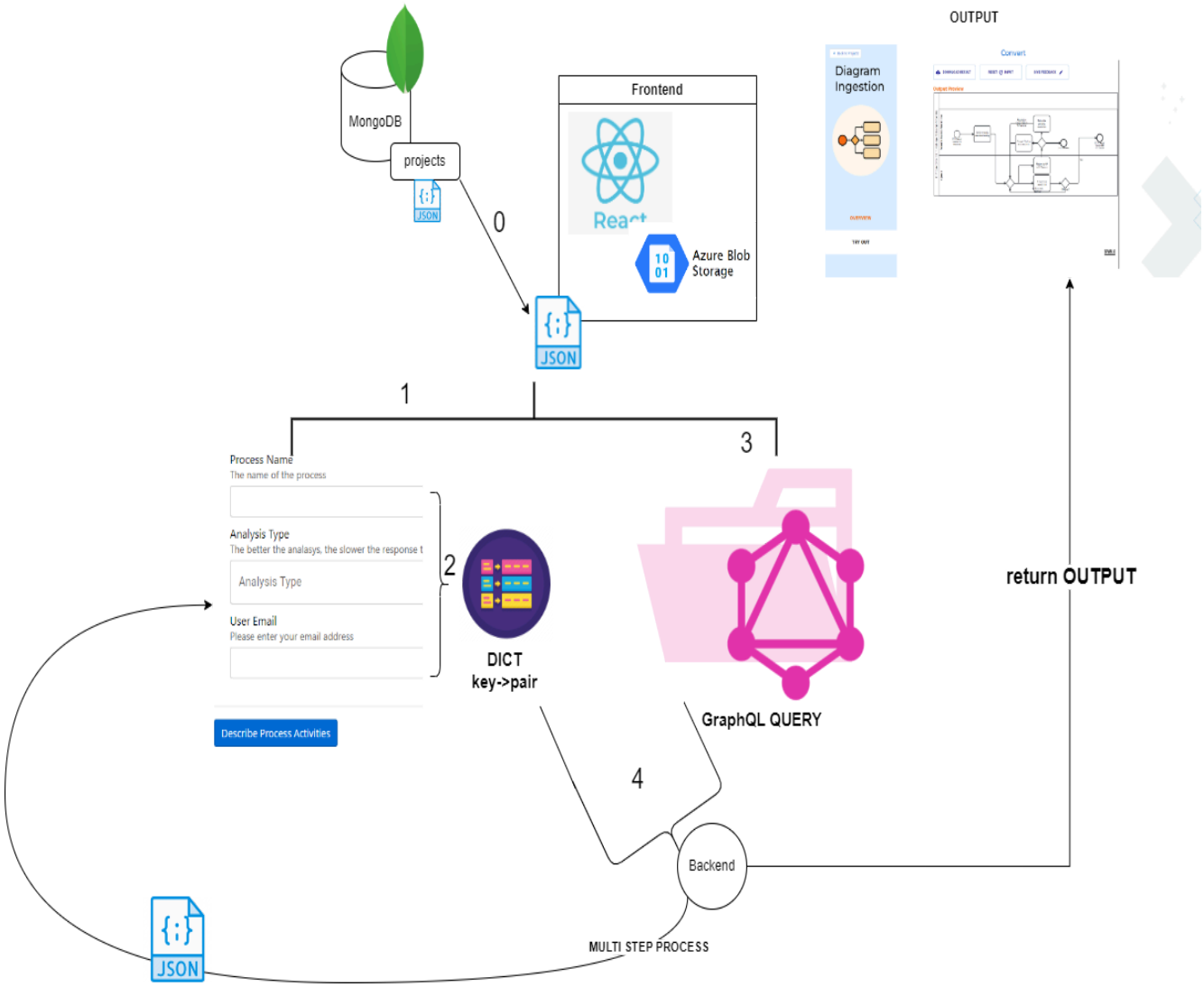
Architectural Design

Architecture

The architecture is divided into 3 main sections:

Frontend



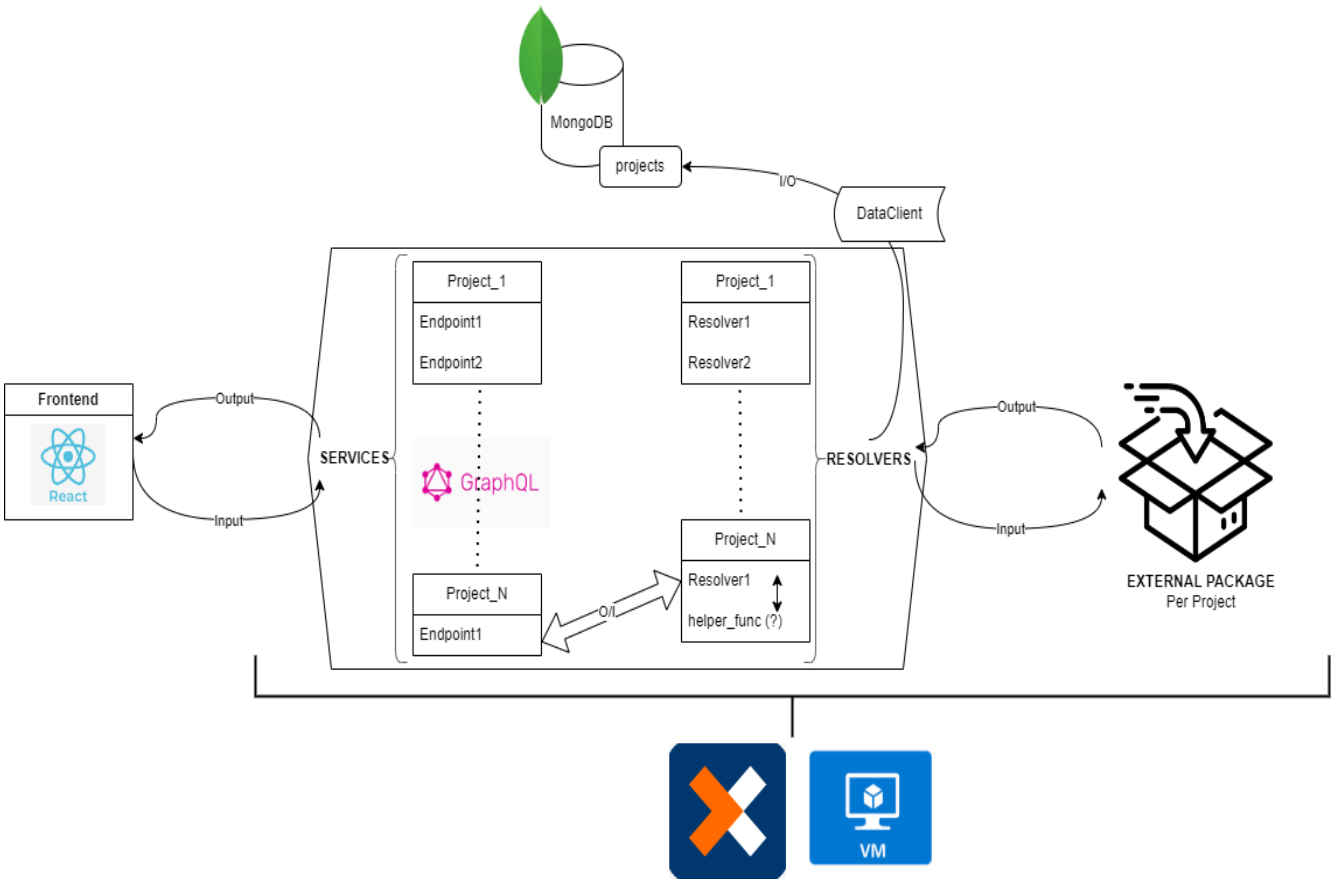
- Written in ReactJs, hosted on Azure BLOB Storage.
- On load - the frontend fetches from the backend the full `projects` collection in the database to display in project library, and render input forms for each project

- The frontend generates the input form using a component that recursively renders containers and/or input fields, each input type has a component that corresponds to it, which all save and handle their values with a global dictionary, the key is the input title, and the value is the form input value
  - (this means that an input called “Process Name”, will be saved in the dict as {ProcessName: “str”})
  - **files** sent this way are converted to base64, and sent as a long string to the backend
- The frontend sends to the backend requests using GraphQL, dynamically generating a query based on the JSON schema for that project

```
const endpointQuery = `
  ${schemaType} ${funcName} {
    ${funcName}(${variables}) {
      response
      preview
      previewFormat
    }
  }
`;
```

- The frontend then takes the result and renders it on the screen. either as a new input form, or as output
- When submitting feedback, the frontend sends the feedback together with the requestID to be appended to the database result entry
- When downloading output, the frontend queries the output for the requestID, and decodes it to be downloaded on user device

Backend



The backend consists of 3 main parts:

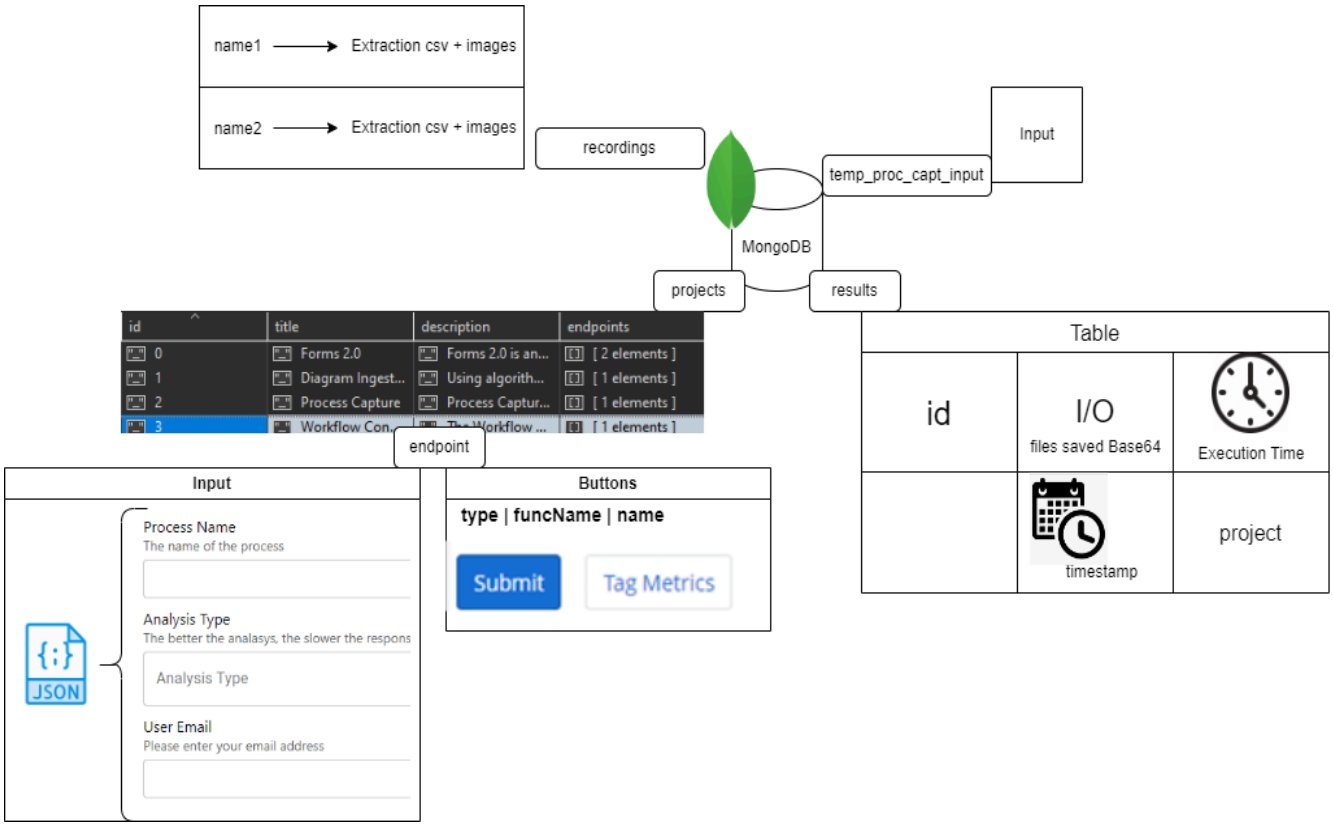
1. **Services**
   - The service gets the GraphQL endpoints, and it (ideally) sends the input to the resolver to be handled.
   - For each project in the lab there is currently a service folder to handle that project’s requests

- Some complex projects, which need multiple steps of input manipulation before being sent to the project function, have a service which calls multiple resolvers to adapt the data to be suitable for the project's function

2. **Resolvers**
   - Each project has it's own resolver folder, in it the resolver file can be found, and contains all resolver function that are called from the service, and occasionally some helper functions called internally.
   - Each resolver handles extracting relevant input, sending it correctly to the Project's API, getting a response, and uploading I/O to the database.
3. **DataClient:** The data client is the instance of the MongoDB for the Lab. It has all relevant upload, update, and fetch functions for interacting with the Database.

- The backend accesses the individual project logic via external libraries installed with poetry
- The backend is hosted on a Nintex VM

Database



The database for the GenAI Lab is hosted on MongoDB and is crucial to the Lab's architecture. It has two primary collections — **Projects and Results** — . Additionally, there are supplementary collections created to address specific challenges, such as caching and input storage for complex projects.

Projects Collection

The `Projects` collection stores all the projects in the Lab. Each project is represented by a JSON document that includes It's ID, name, and other frontend information to render…

The most critical part of the project JSON is the `InputSchema`, which describes the input form and buttons for each project. This schema includes:

- **Buttons:** An array of buttons at the bottom of the form, each described by:
  - **type** - There are 2 types of buttons:

        - *Output:* These buttons trigger the generation of final outputs based on the user's input, providing them with tangible results or insights.
        - *Manipulation:* These buttons allow multiple step interactions by dynamically creating new input forms based on user inputs. This feature allows for adaptable and context-specific input requests, giving more flexibility and accuracy to the developers of a project
    - **funcName** - The name of the GraphQL service endpoint it accesses in the backend.
    - **name** - The name displayed on the button. Releveant only in Manipulation.
    - **single** - Specifies if this action can be invoked multiple times. Releveant only in Manipulation.
- **input** - This is a section on the page where users can add input, information, or upload documents.
    - This input is built out of 2 types: **Objects and Input Fields:**
        - **Objects:** Containers that organize sections of the input form. Each object has a `properties` field that includes all the fields and/or objects inside it.
        - **Invisible Objects:** Hardcoded input values not shown to the user, useful for multi-step processes where certain data needs to be passed to subsequent steps without user input.
        - **Input Fields:** The actual fields seen by the user, each with a specific type, title, and description. Some fields have additional properties like `choices` for multiple-choice inputs.

Results Collection

The `Results` collection captures the output of each user interaction with the Lab. Each entry in this collection includes:

- **Project:** The name of the project to which this result belongs.
- **RequestID:** A unique identifier for the request.
- **File:** The base64-encoded output file, if applicable (for multi-step processes, this is the intermediate output).
- **Filename:** The name of the file when downloaded from the frontend.
- **Input:** A JSON object representing all inputs provided by the user for this entry. For example:

```
"input": {
    "RecordingFile": "BASE64FILE",
    "ActionsCount": 26,
    "RoleName": "Manager",
    "ProcessName": "change details",
    "DeepAnalysis": false,
    "UserEmail": null,
    "RelevantActions": "BASE64FILE"
}
```

- **Timestamp:** The time of upload.
- **Feedback:** If provided, contains `feedbackType` (like/dislike), `description`, and `email`.
- **Step:** Indicates the step of a multi-step process this entry corresponds to, helping determine the output file type.
- **InferenceTime:** A dictionary recording the execution times of different functions, including `TotalTime`.

Supplementary Collections

To address specific needs with the Process Capture project, additional collections have been created:

recordings Collection:

- Serves as a cache for SQLite recording extractions (to VSC files and images) to save time during runtime.
- This helps in avoiding repeated OCR processing by storing intermediate results.

temp_proc_capt_input Collection:

- Created to upload input independently and before generating the output for the Process Capture project.

- This collection is particularly useful when the current machine returns a timeout error, terminating the process midway. Inputs saved here can be processed locally later and returned to the user.