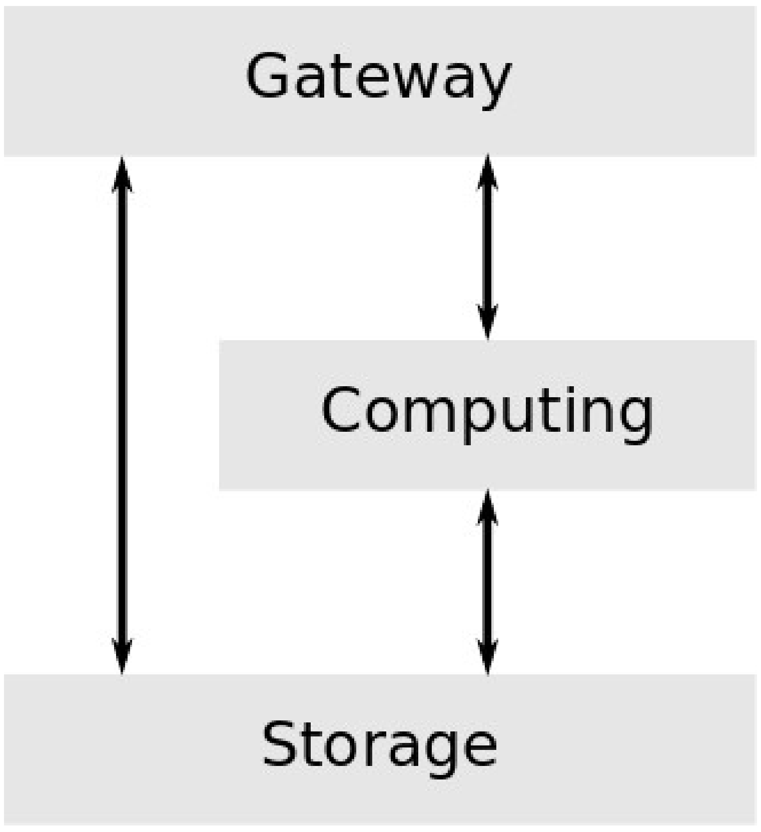
Takeoffs.io Processing Architecture Proposal

The technology behind Takeoffs.io data processing pipeline is detailed on this document.

The system consists of a three layered architecture – a Gateway Layer, a Computing Layer and a Storage Layer. Each layer might consist of several components, and each component is as uncoupled as possible, allowing for scalability and extensibility. The communication between each component is made using RPC and Event-Based asynchronous calls. This architecture resembles a distributed analytics pipeline.

The Gateway Layer acts as an entrypoint for the system. It receives HTTP calls from external clients, therefore acting like a Web Service exposing a REST-like API. Internally, this layer will forward those calls to the subsequent layers accordingly. One of the benefits of this architecture is the possibility to access the Storage Layer directly, retrieving computed data.



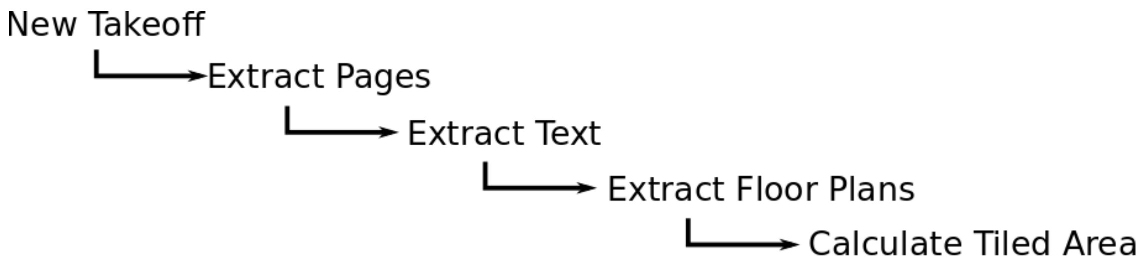
Processing and insights are performed on the Computing Layer. Since the system is expected to be as asynchronous as possible, every computing component is expected to retrieve and store data from the Storage Layer in a way that the Gateway Layer can later retrieve computed data and display to the end user.

Each computing component should be as fine-grained as possible, performing steps that are recognizable by a human whenever possible. Each step is expected to build up sequentially, resulting in a desirable response. As soon as a step finishes its processing, it should store the computed data and raise an event, broadcasting that it has finished. This allows for another component to pick up from there, if the system is expected to keep going on its own, rather than proceeding step-by-step while being watched by a human.

As suggested, each component / step in the Computing Layer should store its data in a way that subsequent components can process further. But also, it is important that computing components also store data in a form that a human can audit and correct if necessary. It is expected that each computing component performs an operation that a human would somewhat trivially perform in order to accomplish the system's objective, and this is the approach taken when selecting the behavior of most steps in the processing pipeline.

Finally, the Storage Layer is responsible for storing computed data in a reliable way. It is also used to store status data (such as what step is currently being performed for a given takeoff). As discussed, data is stored in a way that a human can later audit and/or modify. Therefore, data is stored in a way that should require as little transformation as possible in order to be presented and changed by an auditor.

Currently, the system has only one, proof of concept objective: calculating tiled floor area on a given plan file. The pipeline for computing that using the aforementioned architecture is shown below:



Each step, except "Extract Pages" and "Extract Text" can be straightforwardly performed by a human. "Extract Pages" is very computational and could be thought as being part of the "New Takeoff" step, while "Extract Text" is merely a metadata retrieval step, used to help subsequent steps to have semantic information.

The "Extract Floor Plan" step on the other hand can easily be audited and corrected: a human could see bounding boxes around what the algorithm thinks are floor plans, and either remove it or add a new one. This step being performed wrongly by the algorithm can either mean the previous steps have failed providing enough information (for example, the text extraction failed to detect some important text), or the floor plan detection itself is not working well. In either case, an engineer should be able to compare what has been modified by the auditor and decide whether the algorithm needs fine tuning and how to proceed with that.

To illustrate the desired process, this use case is provided:

1. The user enters the website and uploads a new PDF file containing the plans.
2. The system starts the pipeline, allowing employees to perform live auditing of each step as the pipeline is executed if desired – the employee should be able to stop or pause the pipeline at any moment.
3. The employee should be able to edit the result of any human auditable steps, expecting a more accurate result.
4. When the system finishes all necessary computing steps, an alert is sent to an employee, which proceeds to perform final auditing.
5. The result is sent to the user.

The above use case assumes a system that is already fully automated, which is the ideal scenario. However, to achieve that, from a software engineering and machine learning perspective, an incremental approach needs to take place. Therefore, initially, the system would look like a machine-assisted workflow. As the software progresses, the system will become more and more automated.

To illustrate the machine-assisted workflow, this use case is provided:

1. The user enters the website and uploads a new PDF file containing the plans.
2. The system performs some initial processing and sends an alert to an employee.
3. The employee performs all the expected steps to get the final result. Some of those steps will be automated and the employee is then expected to correct them if necessary.
4. The result is sent to the user.

API Specification

Here follows the API Specification for the above pipeline:

Endpoint	Method	Schema

/upload_file	POST	`{`
		`    'content': base64 binary`
		`    'filename': string`
		`}`
		
		Response:
		
		`{'takeoff_id': string}`

/status/<takeoff_id>	GET	Response:
		
		`[`
		`    {`
		`        step_name: {`
		`            'loading': bool,`
		`            'loaded': bool,`
		`            'message': string`
		`        }`
		`    }`
		`]`

/status/<takeoff_id>/floor_plans	GET	Response:
		
		`[`
		`    {`
		`        'page_number': int,`
		`        'page_data': base64 bytes,`
		`        'bboxes': [[int]]`
		`    }`
		`]`
		
		bboxes are in the form *`[x, y, w, h]`*

/status/<takeoff_id>/floor_plans	PUT	Same as GET, with edited data.

/status/<takeoff_id>/tiled_areas	GET	Response:
		
		`[`
		`    {`
		`        'plan': base64 bytes,`
		`        'tiled_mask': base64 bytes`
		`    }`
		`]`
		
		Tiled mask is a binary mask. Each pixel will hold true if it corresponds to a detected tiled area, and false otherwise.

/status/<takeoff_id>/tiled_areas	PUT	Same as GET, with edited data.