An Interactive End-To-End Machine Learning Platform

Implementation Report

April 25, 2021

Requirements Specification	**Murat Kurnaz**
Design	**Mustafa Enes Batur**
Implementation	**Ömer Erdinç Yağmurlu**
QA / Testing	**Tarek Gaddour**
Final	**Atalay Donat**

Contents

1 Tools

1.1 VS Code

Code Editor

1.2 PlantUML

Charting software

1.3 MikTEX

Latex compiler and package manager

1.4 npm

node package manager

1.5 create-react-app

an intuitive command line scaffolding application easing the development of react applications

1.6 Pipenv

package manager for Python

1.7 Pyenv

virtual environment manager for Python

2 Challenges

2.1 Client

2.1.1 Testing Clients

The front-end is written in React and is composed of presentational components (components), stateful components (containers) and hooks. In separating presentational and stateful components from one another we wanted to ease testing. Although testing presentational components took place without a problem using Jest.js snapshots, render tests and some simple consistency tests, stateful containers were harder to test, since they required extensive mocking of React's hook and lifecycle events.

2.1.2 Sensor Data Collection

Browser Inconsistencies

from https://github.com/PSE-TECO-2020-TEAM1/client/issues/5#issuecomment-817308653

There are two main APIs on sensor access in browsers right now, namely the Sensors API, which is incorporated into the web standard and supported by 71.03% of all users worldwide.

On the other hand there is the legacy DeviceMotionEvent API, which was an experimental technology designed before the aforementioned Sensors API was drafted. It is supported by 94.97% of users worldwide (albeit with handicaps).

In this application, we are using the new standard Sensors API, which is only supported by Chrome and Chromium based browsers like Edge, Brave etc. for now. Apple refuses to implement the new API citing privacy concerns, there is no information on why Firefox doesn't implement it. Since on Apple platforms all browsers from all vendors use the Safari WebView, this new API doesn't work at all on Apple devices.

The DeviceMotionEvent API is unfortunately not suitable for use at all. All three different major browsers (Chrome, Safari and Firefox) have a different understanding of what coordinates they return and have no documentation of which units they return the data in.

We've tried to use a polyfill with the branch sensors-polyfill, but were getting totally different results with different browsers (and with firefox totally broken results. Because of this reason we've decided not to support Apple users at all.

Magnetometer Originally we wanted to support Magnetometer sensor too, since it was (supposedly) supported by the browsers we were targeting (on caniuse.com).

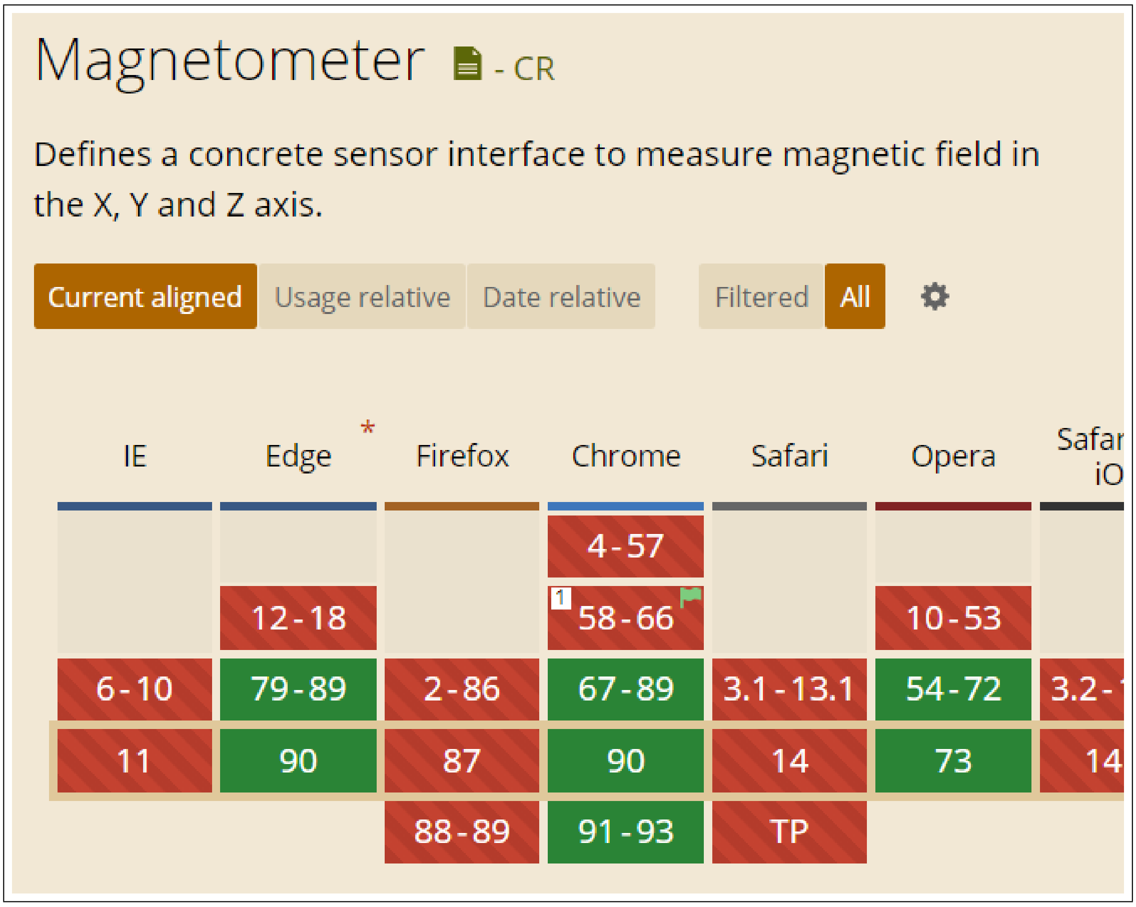


Figure 1: caniuse.com - Magnetometer

After implementing the sensor, however, we've noticed how no matter what we've tried, we weren't getting any data, and after more investigations we've discovered that the 'Magnetometer Sensor' and the 'Magnetometer Sensor API' are two different entities separate from each other, and dropped support for magnetometers.

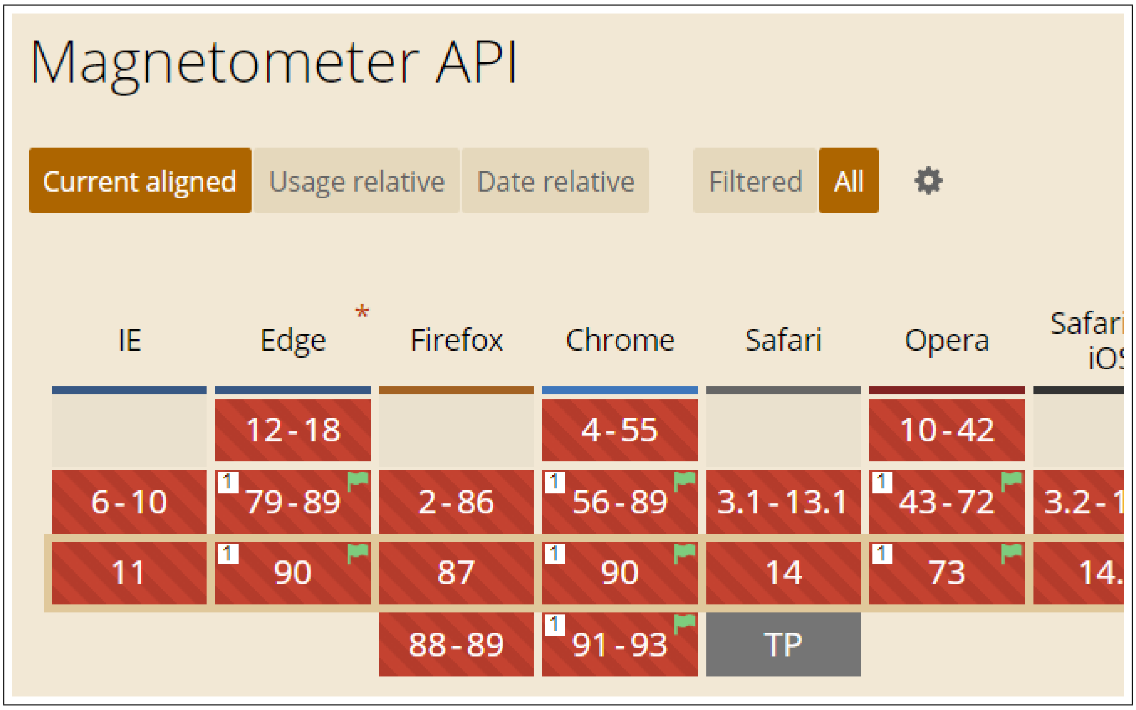


Figure 2: caniuse.com - Magnetometer API

2.2 Workspace Management

Making a Consistent Server and Covering Failure Cases As the workspace management handles storage of user data, it acts as a bridge between the client and the model management. Thus the validity of these data is a very crucial part of the work. Although the client prevents most of the invalid requests, such invalid requests could be handcrafted and sent to the server or the client itself could possibly generate an invalid data by an error so each request needed to be validated. Finding the error cases and handling them generated a lot of work, because finding the not so obvious error cases needed a general analysis of the action caused by the request on the server and possibly on the model management.

Performance-Space Trade-Off Decisions Swaying from the initial design is in most cases undesirable and comes with a cost. However as the implementation phase progressed, it came to light that the initial design fell short on some aspects regarding the functionality (see **??**). The app is designed to be scalable so when introducing new changes to the codebase we had to take performance related issues into consideration and it slowed down the progress when the changes had conflicting aspects regarding performance and space.

2.3 Model Management

This was a very challenging part of the project for us, since we had no experience with the libraries that we use (namely scikit-learn) as well as the programming language Python and the machine learning pipeline. Our first attempt showed that our design for the service was not appropriate for the requirements so we had to iterate twice, which included writing the whole service from stratch.

2.3.1 Understanding Machine Learning Pipeline

During the design phase, we had tried to get a grasp of the machine learning process that we were going to implement but we had no real experience with it. This resulted in a very abstract design since we were unsure how different mechanisms interacted with each other. As we experimented with the frameworks, a series of new requirements came into play. This included changes in other components of the system that Model Management depended on like the front-end clients. As a result, we had to constantly communicate with each other, synchronize our works and give feedback to each other. With time, our better understanding of the machine learning process made things easier and allowed us to solve problems quickly.

2.3.2 Caching Mechanism and MongoDB Document Size Restrictions

The first issue that we have faced was the caching mechanism for the processed data which helped the training with avoiding repetitive calculations when possible. Our first naive approach was to store everything in an array. It became quickly apparent that saving the data this way complicated the code immensely and was very error prone. The solution was to store serialized DataFrame objects as a document in the database. During the testing we realized that MongoDB did not allow for documents larger than 16 MB in size. Another framework (GridFS) that wrapped MongoDB allowed us to solve this problem and store files without any size restrictions.

2.3.3 Multiprocessing and Serving Many Clients in Parallel

The biggest challenge of the Model Management service was allowing multiple training or prediction processes take place on the server in parallel. Because of the computing intensive nature of the machine learning processes, it was not feasiable to run the training or prediction on the same process that handled the client requests which would greatly reduce the server responsiveness. After a through research on possible solutions, we have decided that the best solution was to spawn new processes for each computing intensive task (i.e. training/prediction), since we discovered that multithreading does not work as expected in programming languages that depend on an interpreter. The hardest part was to synchronize the processes and handle the interprocess communication. The solution for data transfer between the server and the child processes was using Unix Pipes and the synchronization problem was solved by using counting semaphores.

2.4 Auth

No notable challenges.

3 Statistics

3.1 client

Lines of code	TBD
Test coverage	TBD
Number of commits	RBD

3.2 workspace-management

3.3 model-management

3.4 auth-management

3.5 Total

4 Changes from Design

4.1 Authentication

No notable changes.

4.2 Front-end Clients

4.2.1 Single Codebase

In the design document, we'd envisioned two different codebases for the different edge (mobile) and management (desktop) clients. During development, however, it has become obvious that a single codebase with client side routing and bundle separation using tree shaking was more suitable for our application. We are bundling both applications in a single router, and there is no clear separation between each client. During development, special care was given to reduce cross dependencies between both parts to a minimum in order to reduce the bundle size for edge devices.

4.2.2 /lib folder

Originally, there were only two auxiliary classes in the whole client (MobileAPI and DesktopAPI), while everything else was a React component. During development, we made use of custom react hooks (/lib/hooks) to refactor common stateful logic into reusable parts. Apart from hooks, sensor data collection needed it's own abstraction over the clunky Web API implementation, which we've placed in /lib/sensors.

4.2.3 API endpoints

In order to accomodate changes in the backend, both the Mobile- and DesktopAPI have undergone major changes in the interfaces they implement.

4.2.4 Component Library

In the design document, we'd specified Material-UI as the component library that we were going to use. During development we've found it too clunky, heavy and hard to develop with and replaced it with the Evergreen Component Library from segment.io. This component library also comes with it's own opinionated 'CSS-in-JS' styling library, through which we were able to style the application with a rapid pace.

4.2.5 ModelOptions Component

This component faced the most changes during implementation. We'd decided to distribute some model creation options to the sensor-components themselves instead of using the same set of parameters for each component. This had it's implications on the client and it had to be updated accordingly.

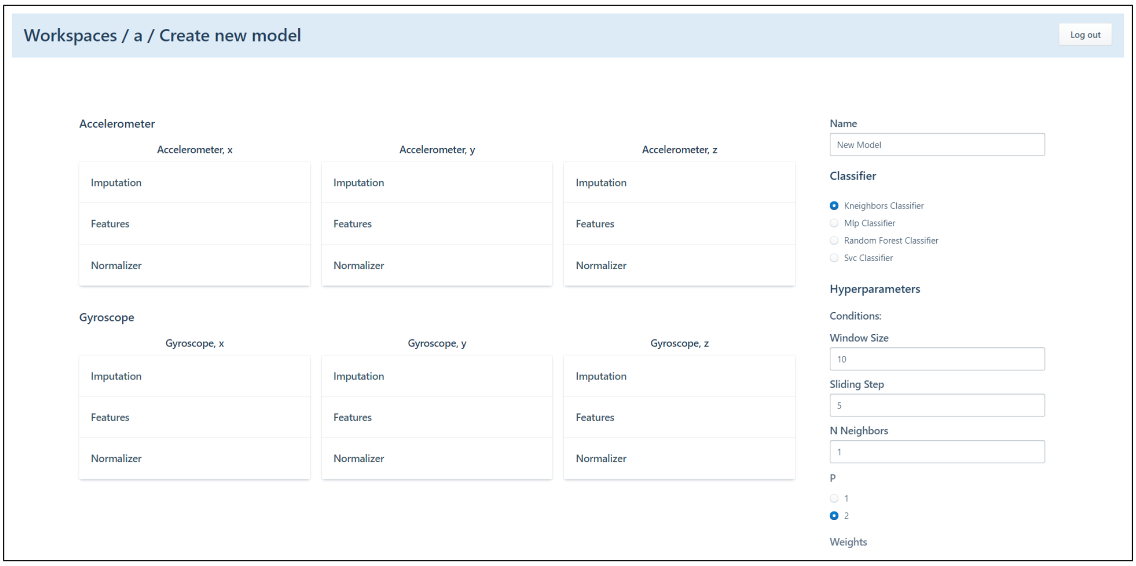


Figure 3: New ModelOptions

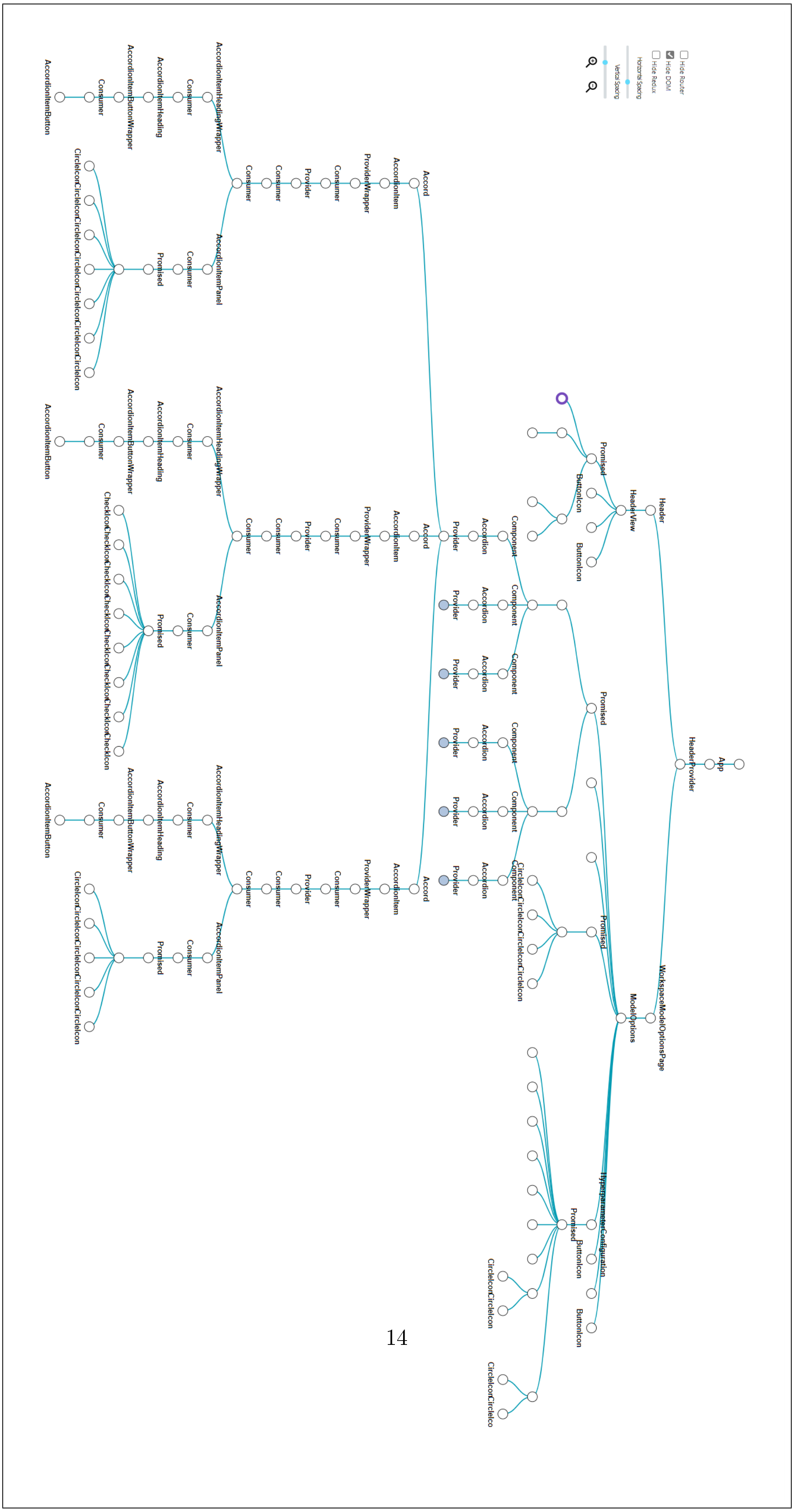


Figure 4: New ModelOptions Component Graph

4.3 Workspace-Management

4.3.1 Label

Label isn't saved as an embedded document anymore. To achieve the same functionality label now keeps the workspaceId of the workspace it belongs to and the number of samples it is the label of.

4.3.2 Data Point

ObjectId is now omitted as it is not used and causes disarray in the corresponding responses.

4.3.3 Sensor Data Point

ObjectId is omitted here as well on the same grounds. In addition name of the sensor is also added to schema to simplify communication with the client.

4.3.4 Sample

Sample does not embed the label data anymore, instead it saves the id of the label. Implementation of setTimeFrames method is also delegated to the workspaceController.

4.3.5 Workspace

SubmissionId SubmissionId is not saved as a simple string anymore, instead it has its own interface with the field hash as string. This change was done to provide flexibility for possible future use cases, such as making submissionId expire after a while etc.

Further Additions Workspace now holds lastModifiedDate to enable model management to cache the machine learning pipeline, sample- and labelIds to speed up database queries/checks.

4.3.6 Routes

General Changes Added new bad request responses that explains the error user is getting.

GET /api/sensors Removed as it didn't provide any utility other methods didn't cover.

GET /api/workspaces/workspaceId/samples Query parameters are changed to showDataPoints and onlyDate to be more intuitive.

GET /api/workspaces/workspaceId/samples/sampleId Response now also includes the timeframes of the sample.

PUT /api/workspaces/workspaceId/samples/sampleId/relabel labelName is now passed in the query instead of labelId.

GET /api/workspaces/workspaceId/labels Response body now also includes the sample count of the label.

POST /api/workspaces/workspaceId/labels/create labelName is now passed in the body instead of the query.

PUT /api/workspaces/workspaceId/labels/labelId/rename labelName is now passed in the body instead of the query.

PUT /api/workspaces/workspaceId/labels/labelId/describe description is now passed in the body instead of the query.

GET /api/workspaces/workspaceId/submissionId This route has been changed to GET /api/workspaces/workspaceId/generateSubmissionId to make its function clear.

GET /api/submissionConfig Label objects now include their labelId in the response body.

POST /api/submitSample submissionId is now passed in the request body.

4.4 Model Management

As described in the challenges section, we had to redesign this service. The main reason is the multiprocessing needs of the application that was unforeseeable for us during the design phase. The complete changelog requires a completely new design document but the following are the most important changes.

4.4.1 Router

- Split the Router class into two classes, CommonRoutes and WorkspaceRoutes.
  Reason: API endpoints that need authentication are all under a specific workspace, so the split made it easier to handle the authentication of requests. We use an authentication middleware for all workspace routes.

- Removed generatePredictionId method
  Reason: We delegate this responsibility to the MongoDB client that generates a new ID for each document that is inserted into the database.

- Removed the trainers and predictors fields
  Reason: Trainers map was planned to be used to track the progress. Since each trainer starts in a new process, this was not possible and we used the database for progress tracking instead. Predictors are in a new class PredictionManager described below.

4.4.2 Request and Response Classes

- Renamed the data classes.
  Reason: The classes now represent the actual content instead of the name of the API endpoint (i.e. TrainReq -> TrainingConfig)

- Seperate (sometimes duplicate) classes for domain models and API models
  Reason: The first iteration of the service showed us that using the same models for both internal logic and API endpoints are problematic because of the possible changes to the endpoints. By seperating the classes, we had the flexibility to change endpoints/parameters without affecting the logic code, since the API models are converted to domain models.

- Added data validation for endpoints
  Reason: During the design, the data validation was missing. We have added the

data validation for models with comprehensible error messages that frontend shows the end users.

4.4.3 Database

- Added wrapper classes for database queries
  Reason: At first we were calling the functions of PyMongo directly in the application code. We thought this was not a good idea as it was not possible to use a different database. With that in mind, we implemented several wrapper classes which implement database queries.

4.4.4 Database Models

- Added (de)serialization methods for database models
  Reason: We save large data in the database in binary form. (bytes type). To prevent errors during the deserialization because of the unknown type of the binary objects, we added methods that complete the type information for each binary document to database model classes.

4.4.5 Training

- Training a model is handled by a new process
  Reason: The training of a model takes a notable amount of time, especially if there are a lot of samples. If the main process were to train the model, any requests during the training could only be handled after the training is finished. This was a terrible idea, so a new process is created to handle the training and the main thread is then able to handle other requests during the training.

- Added DataSetManager and TrainingManager classes
  Reason: The Trainer class which we initially designed was a typical God object in which the training pipeline, all the training algorithms and the database queries are implemented. With seperation of concerns in mind, we decided to split this Trainer class in three classes: DataSetManager handles the database queries, the Trainer class has all the algorithms implemented and the TrainingManager handles the training pipeline.

4.4.6 Prediction

- Prediction is handled by a new process
  Reason: This has the same motivation with handling the training with a new process: Being able to handle other requests while a prediction is under way.

- Added DataSetManager and PredictionManager classes
  Reason: This also has the same reasoning with the training: The Predictor class we had was a typical God object. The DataSetManager handles the database queries, the Predictor class implements all the algorithms and the PredictionManager handles the prediction pipeline.

5 Functional Requirements Coverage

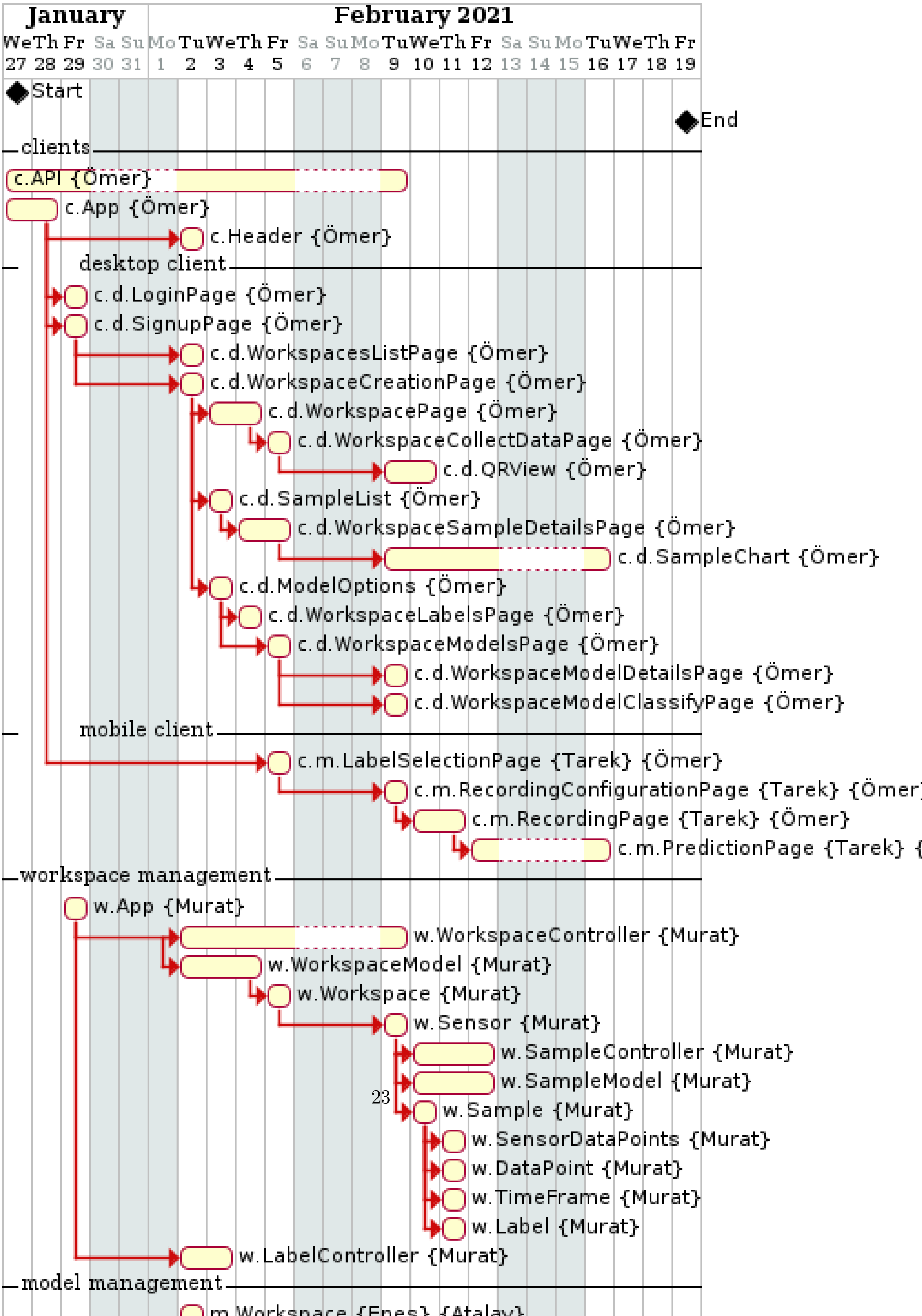
5.1 Mandatory Requirements

Number	Requirement Name	Implemented?	Notes
NUM	Here comes the name	YESNOPARTIALLY	

5.2 Optional Requirements

Number	Requirement Name	Implemented?	Notes
NUM	Here comes the name	YESNOPARTIALLY	

6 Planned Schedule



7 Actual Schedule

TBD: CAN EVERYONE UPDATE THE GANNT CHART IN THE PLAN FOLDER WITH WHAT THEY'VE DONE