Goal Model

Version Control

Version	Date	Update notes
v1	26 Mar 2021	• Initial write-up of requirements based on document provided from clients • Minimal changes from client specification
v2	29 Mar 2021	• Added table view specifications and fixed formatting after team feedback
v3	04 Apr 2021	• Added goal model diagram based on Do-Be-Feel list
v4	11 Apr 2021	• Adjusted Do-Be-Feel list and goal model diagram based on client meeting
v5	19 Apr 2021	• Adjusted Do-Be-Feel list and goal model diagram based on client meeting • Main changes ◦ Removed the data visualisation requirements ◦ Formalised what's to be persisted
v6 (Current)	30 Apr 2021	• Added paragraph explaining the reason for using two goal models

The project consists of two independent deliverables: the predictive monitoring module and the training module. The two will be deployed as separate instances and work independently of each other. Given that the modules we're developing work independently we decided to create two goal models to both reflect the distinction between the modules and improve the readability of the individual models.

Predictive monitoring module

Who - Do - Be - Feel

Using MoSCoW Prioritization: ***Must*** *>* ***Should*** *>* ***Could*** *>* ***Will Not Have***

Who	Do	Be	Feel
Operations manager	**Must:** Import parquet formatted input logs into the Apromore predictive monitoring plugin	**Must:** Scalable, i.e. capable of making predictions on tens of thousands of cases and training models on logs of up to one hundred million cases	**Should:** Fresh and professional
Business analyst	**Must:** Be able to view predictions in a table view (e.g. one caseId per row)	**Must:** Robust, i.e. capable of recovering from sudden crashes and picking up work where it left it.	
	Must: Be able to export predictions for further analysis (e.g. via CSV files)	**Must:** Concurrent, i.e. capable of handling dozens of concurrent requests.	
	Must: Be able to view aggregated prediction data such as average case length, average case duration, completed case count etc.	**Must:** Asynchronous, i.e. the backend microservice should handle the client's request for predictions in an asynchronous manner	
	Must: Be able to generate dashboards containing predictions of two basic types using provided pickle files: • time remaining till process completion • binary outcome (e.g. next event)	**Must:** Compliant with the Apromore license terms and use open source libraries and packages.	
	Must: Create predictive monitors consisting of an arbitrarily large number of predictors	**Must:** Thoroughly tested: Unit tests & integration test coverage needs to be at least 70%.	

	Must: Be able to generate a case-level dashboard from logs containing open (incomplete) cases. The case level dashboard will consist of the following fields: • case ID • relevant case attributes specified by user • most revent event that occured in the case • number of events that occurred in the case • outcome predictor ◦ for time predictors: remaining time ◦ for categorical predictors: expected outcome and its probability ◦ for next event predictors: next predicted event (including case complete event) and its probability	**Must:** Built using a decoupled architecture with • a backend microservice implementing the core business logic written in Pythonu using ~~Flask or~~ FastAPI • A frontend client built using the Angular framework	
	Must: Persist the user created monitors in the server file system.	**Must:** Using a FIFO request queueing system (e.g. RabbitMQ) to asynchronously handle requests. When queried by the client the backend should be able to check the status of a given predictive report generation job, and retrieve the results once they are ready.	
	(new) Must: Be able to validate that the sets of files uploaded by the user (event logs, schemas, filters, predictors) are compatible with each other (i.e. have follow the same schema)	**Must:** Built using a master-slave architecture for the backend. A master node should manage the request queue and distribute tasks among a scalable number of worker nodes.	
	Should: Be able to cancel dashboard creation task that's in a queue or being processed	**Must:** Use a RESTful API to wrap around the predictor service.	
	Should: Be able to generate predictions using pickle files created using the Training module	**Must:** Containerised using Docker using a single Docker compose file	
	~~**Should**~~ **Could:** Sort the table view by field (i.e. by column)	**Must:** ~~Stateless~~Relatively stateless (user created monitors are persisted)	
	~~**Could**~~ **Will Not Have:** Be able to persist the prediction outcomes into a database (e.g. PostgreSQL / MySql)	**Must:** Well documented, especially with regard to its API (e.g. using Swagger)	
	Will Not Have: Be able to analyse real-time inputs generated through a stream of events (using technologies like Kafka)		
	~~**Should**~~ **Will Not Have:** Be able to visualise prediction statistics (e.g. histogram of remaining time, distribution of classes in categorical predictions etc)		

Training module

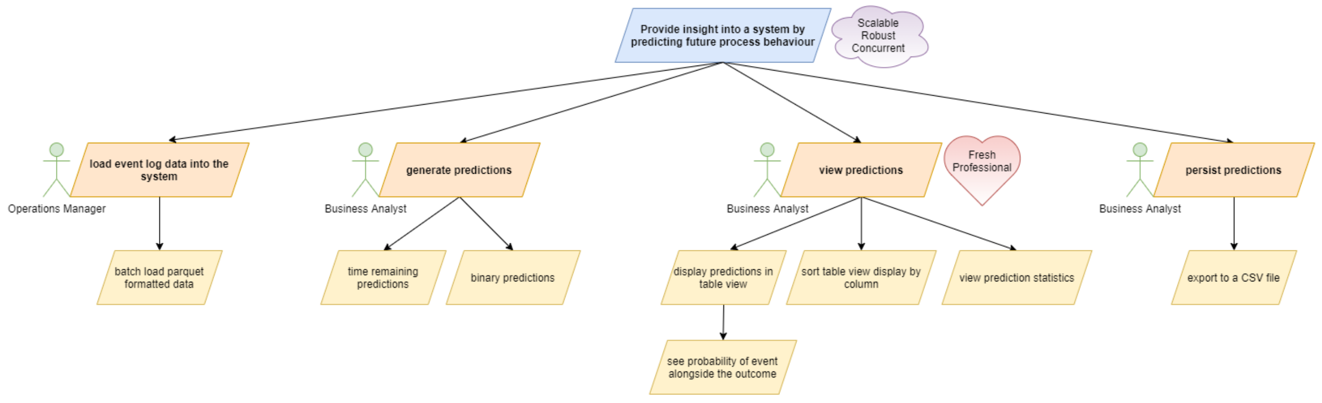
Who - Do - Be - Feel

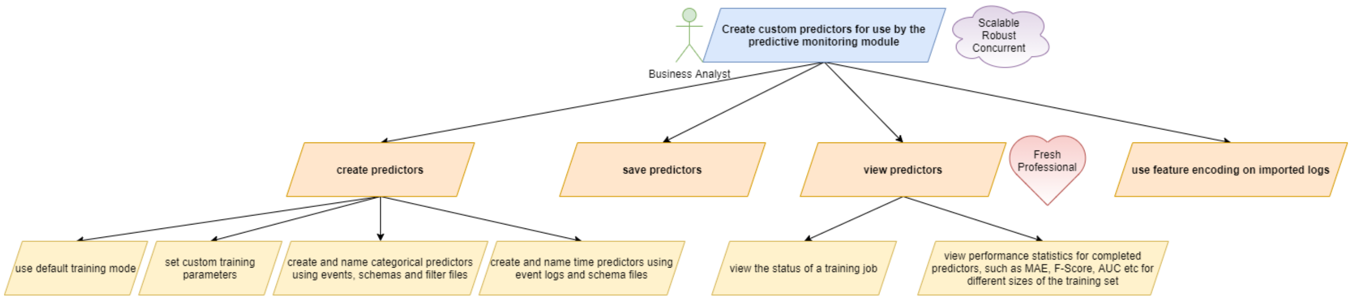
Using MoSCoW Prioritization: ***Must*** *>* ***Should*** *>* ***Could*** *>* ***Will Not Have***

Who	Do	Be	Feel
Business analyst	**Must:** Be able to create a remaining-time predictor provided • an event log containing completed cases • the matching schema file uploaded by the user and assign it a name. (reworded for clarity)	**Must:** Built using a decoupled architecture with • a backend microservice implementing the core business logic written in Python using FastAPI • A frontend client built using the Angular framework	**Should:** Fresh and professional
	Must: Be able to create a binary outcome predictor provided • an event log containing completed cases • the matching schema file • the filter file defining classes to be detected uploaded by the user and assign it a name. (reworded for clarity)	**Must:** Robust, i.e. capable of recovering from sudden crashes and picking up work where it left it.	
	Must: Be able to use default training mode to create predictors • Encoding: Frequency • Bucketing method: None • Prediction method: XGBoost	**Must:** Concurrent, i.e. capable of handling dozens of concurrent requests.	

	Must: Display pertinent statistics describing the performance of the predictor of a given predictor (such as the F-score and the AUC) for different prefix lengths (different number of events). (reworded for clarity)	**Must:** Asynchronous, i.e. the backend microservice should handle the client's request for predictions in an asynchronous manner.	
	~~**Should**~~ **Must:** Store completed predictor training jobs and display those in a table.	**Must:** Built using a decoupled architecture with • a backend microservice implementing the core business logic written in Pythonu using ~~Flask or~~ FastAPI • A frontend client built using the Angular framework	
	Should: Support feature encoding on imported logs, e.g. aggregation encoding, index encoding etc.	**Must:** Using a FIFO request queueing system (~~e.g.~~ RabbitMQ) to asynchronously handle requests. When queried by the client the backend should be able to check th e status of a given predictive report generation job, and retrieve the results once they are ready.	
	Should: Be able use advanced mode for creating predictors, i.e. set custom training parameters.	**Must:** Built using a master-slave architecture for the backend. A master node should manage the request queue and distribute tasks among a scalable number of worker nodes.	
	~~**Must**~~ **Will Not Have:** Be able to create a chart showing the mean absolute error on the training set for a given remaining-time predictor, for different prefix lengths (different number of events).	**Must:** Compliant with the Apromore license terms and use open source libraries and packages.	
		Must: Thoroughly tested: Unit tests & integration test coverage needs to be at least 70%.	
		Must: Containerised using Docker	
		Must: Well documented, especially with regard to its API (e.g. using Swagger)	

Goal model


Provide insight into a system by predicting future process behaviour
Scalable
Robust
Concurrent
Operations Manager
load event log data into the system
batch load parquet formatted data
Business Analyst
generate predictions
time remaining predictions
binary predictions
Business Analyst
view predictions
Fresh
Professional
display predictions in table view
see probability of event alongside the outcome
sort table view display by column
view prediction statistics
Business Analyst
persist predictions
export to a CSV file


Business Analyst
Create custom predictors for use by the predictive monitoring module
Scalable
Robust
Concurrent
create predictors
use default training mode
set custom training parameters
create and name categorical predictors using events, schemas and filter files
create and name time predictors using event logs and schema files
save predictors
view predictors
Fresh
Professional
view the status of a training job
view performance statistics for completed predictors, such as MAE, F-Score, AUC etc for different sizes of the training set
use feature encoding on imported logs