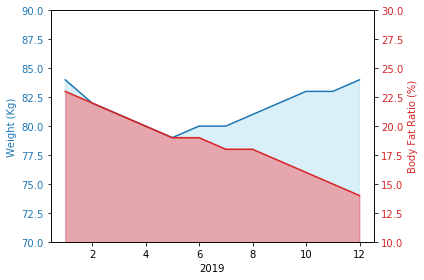
# 4.1 Description

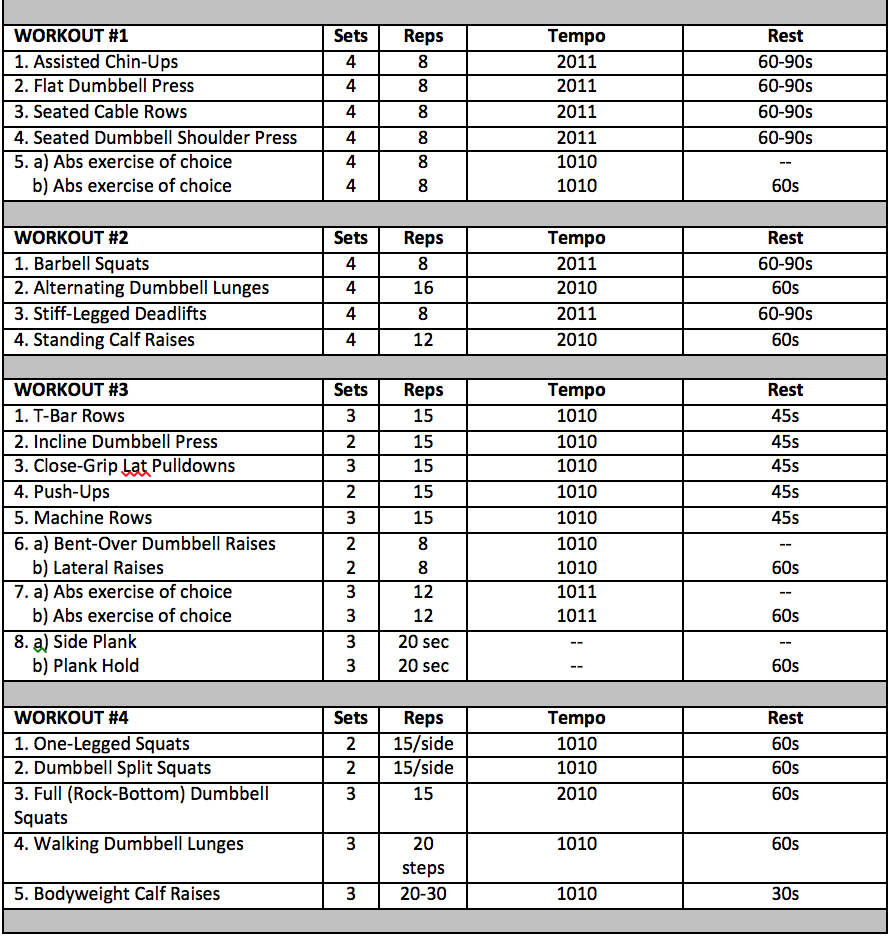
The system developed uses several elements in conjunction to work correctly, the core of the system is the actual AI, which has the name of Sam as it offers a unisex way of talking with the chatbot. In this chapter we will go into detail on how the system has been built and how the different elements provide the support for the different features the bot has.

The objective of this system is to create a simple way of tracking your progress, create dynamic tables which learn from you and provide a correct assessment and guidance to reach the users objective. The following points reflect what a final solution features would be:

- Progress tracking: With this the bot and the user can have an idea of how the training sessions is progressing, based on the results from the users measurements the bot may choose to change the users diet, exercise table or remind the user to follow the diet table given.



- Exercise table creation: The bot provides new tables every month or may change if it sees throughout the month very little progress from the user. This table would be designed to fulfill the objective the user wants.



- Diet table creation: Like the exercise table, this diet table would be based on what objective the user wants to reach. The bot will also consider what allergies the user may have and try to progressively adapt the table in order to transition from the user’s normal diet to the new diet without altering it all at once.

- Training sessions: This is a feature that could be added in the future if the speech technology keeps improving, as it would be ideal to have headphones on while Sam guides the user through the training session, this would make the user use less the phone and still keep track of brake times.

- Gym location: If a gym franchise or a multiple gym service bought the application this feature could make it easier for the user to find a gym nearby or maybe check what gyms are available at a certain location.

- Exercise tutorials: This could be a very useful functionality to make it easier for the user to know how to do an exercise, as some exercises by name may be confusing and this will make it easier to find out, compared to looking for it online.

- Route recommendations: At the moment the core functionality is based on gym training, especially weight lifting, but an interesting functionality that could be added in the future is for outdoor bike and running routes. With recommendations from users on the best routes.

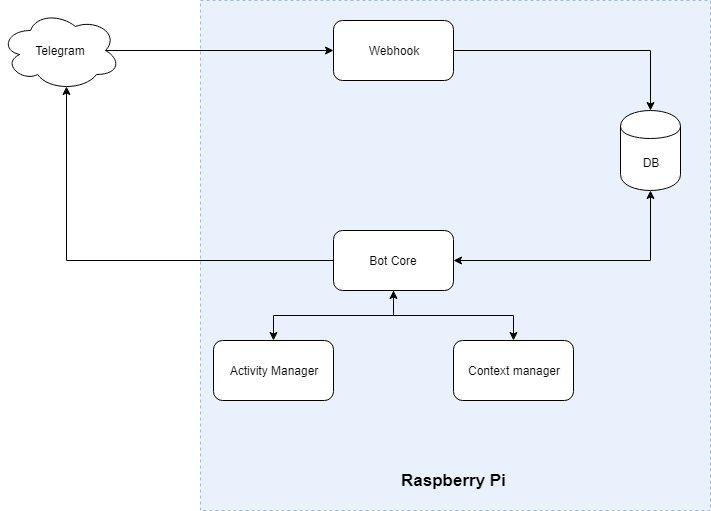
- Team meetings: This functionality focuses on the social part of training, where users can arrange team matches between the users in what interest them more, even if the users interests are outdoor training or cycling teams, where they can arrange a route up a mountain, this will provide a community feeling in the application.

# 4.1 Design

In this project we find that there has been two different designs, the original design was developed in a way to make the system homemade, without depending on an external platform which even though it provides a base where to start developing it limits the control the developer has in adding new functionality.

The second and final design does use an open source platform as the base to start developing, this had to be done because of the complexity related in developing the proprietary solution in relation to the time available to develop it. Further down the memory the reason for changing the design will be explained in more detail.

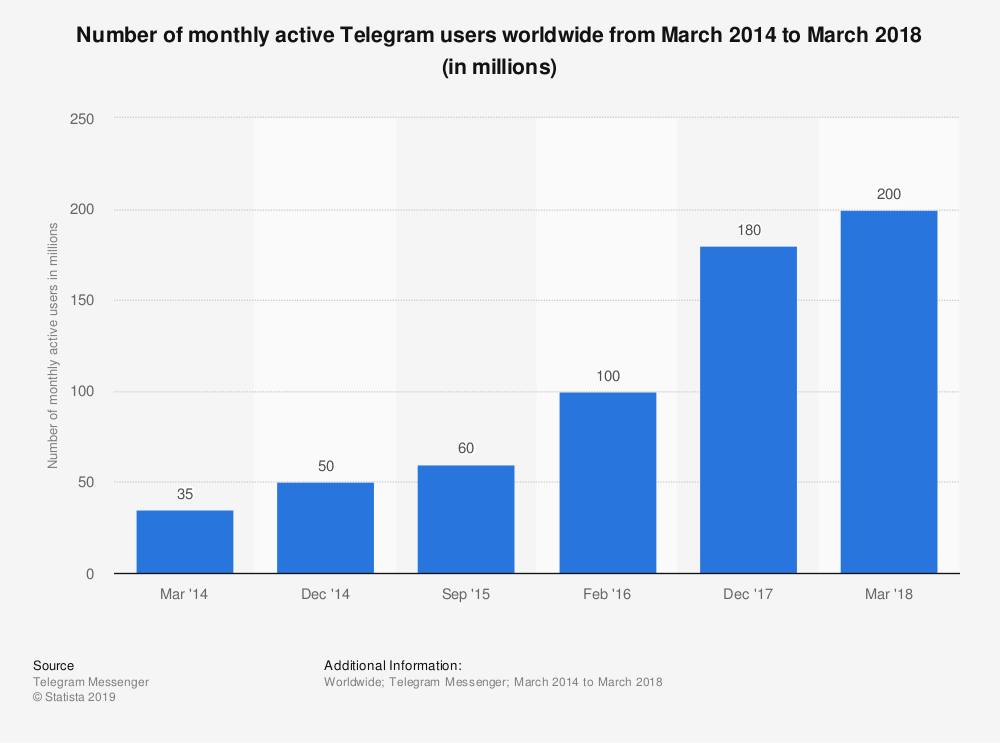
## 4.1.1 Original Design



### 4.1.1.1 Frontend

Following a model view controller architecture, the project uses as the view part Telegram, a chat platform for the users to contact the bot. The bot can be ported to different chat platforms based on preferences from the users, the reason Telegram was chosen instead of another platform where:

* Higher user count: Compared to Slack, another chat platform with support for bots, Telegram had for the year 2018 200 million active users, while Slack only has 10 million. The reason to choose a chat platform with a higher count for active users is because with more users it is easier to penetrate the market. As the graph shows below it is also important that Telegram has been steadily growing in recent years with an increment of 100% in one year from 100 million to 200 million active users.



* API: Telegram has a very well documented API which makes it easier to integrate with the bot, also with previous knowledge in developing with Telegram it was the logical choice to use.
* Webhook: It provides a way of creating a bridge between Telegram servers and the Raspberry Pi to reduce stress at the network as Telegram automatically sends messages to the server instead of relying on the Raspberry Pi polling Telegram servers.

WEBHOOK

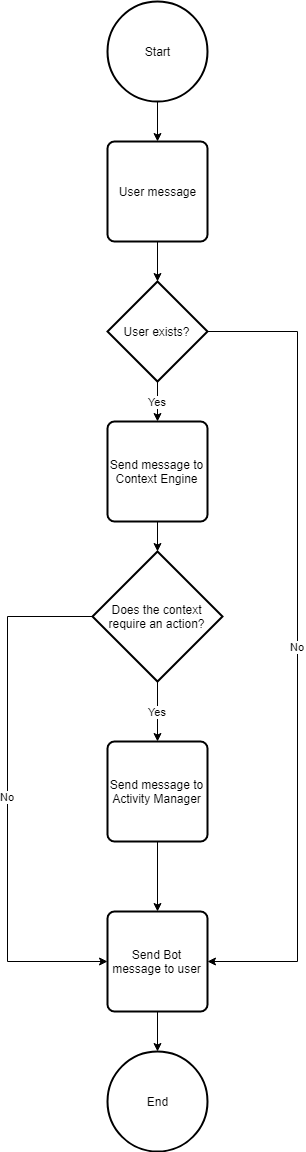
After analyzing options to get the messages from Telegram, it was decided for this design to have a webhook instead of polling the server. A webhook is the bridge for communicating with the Raspberry Pi allowing to relieve the network from redundant traffic. The way a webhook works is like a web server which Telegram calls it whenever a message is received from a user. Whenever the webhook receives the message, it stores it in the database where the bot can check if anything new has arrived, this allows to transfer the stress from the network to the pc which reduces latency when receiving messages as polling a server takes longer than polling a database.

### 4.1.1.2 Backend

As the frontend was mostly managed by Telegram there wasn’t much to design except the webhook to support the message delivery.

For the backend we find the controller and the model for the bot. The way this two were designed is in that the controller was the communicator, were it managed the database where it extracted the user’s data two give back to the functions that performed the tasks. The first thing the controller does when getting a message from the database is authenticate the user, this is done with the telegram id sent from telegram, this id is unique to each user which makes it secure to identity fraud, at least from the applications end.

Below is the original flow for a message sent by the user to the bot, the design is at a high level to see where the message goes through until it gets to the response.

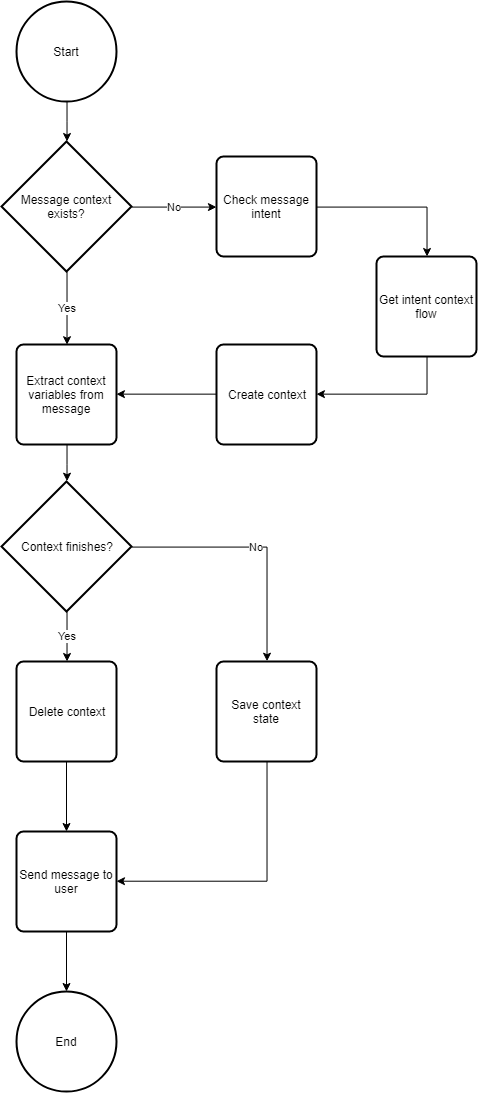


The flow follows starts once the message is grabbed from the database, the first thing is to verify that Sam is talking to a user from the platform, the idea is that the user doesn’t belong to the platform Sam offers to join the club. The complex part of the application was the context engine side of it.

CONTEXT MANAGER

The context engine is an engine which objective is to give the bot context from the user, this gives intelligence to the bot as it can remember what has been talked before. It also has the function of extracting data from what the user’s message. The following flow shows how it is intended to work. These are the main design objectives when creating the context engine.

* Being able to complete actions without following the happy path structure, in other words, if the bot wants to add measurements and there are two different measurement types there isn’t one way to provide those measurements, the user can facilitate them in several ways.
  + I want to measure myself: In this case the bot then will ask for the user’s weight.
  + Sam please add 80 kg as my new measurement and my current body fat ratio is 19%: In this case the context engine will extract both variables and delete the context as all relevant data has been extracted.
  + Sam my new body fat ratio is 19%: to which Sam will detect that the user has said its body fat ratio but not his weight and will ask the user to provide his weight as well.
* Keep the user in track whenever the user deviates from what is being asked.
* The context engine must be capable of extracting data correctly
* Scalability is an important objective that the context engine must be designed for.
* The flows can be generated without having to manage code directly, the idea to make this work is to have the flows standardized as JSON formatted file.

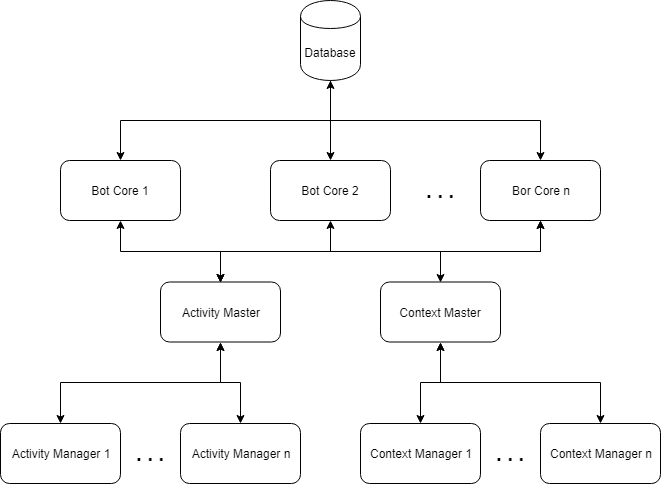


ACTIVITY MANAGER

The activity engine is the one in charge of performing the tasks required for the bot to be functional, the design is simple but the complexity in this manager is the functionality the developer wants Sam to have. The design for this is that it must be simple to add new functionality and not have to change to much code. The idea is to use the flows we talked about in the context manager section, in these flows if the context requires an activity to have the name of such activity in the JSON file and execute it in the activity manager.

SCALABILITY

An important requirement for Sam is for him to be scalable, to reach this objective the architecture is designed in modules so that if a certain part of the application is receiving a higher level of stress to be able to scale it independently. This is done by having two masters, one for the activities and one for the context manager, these when launched by themselves act as a normal manager, but when several are launched acts as a load balancer.



## 4.1.2 Final Design

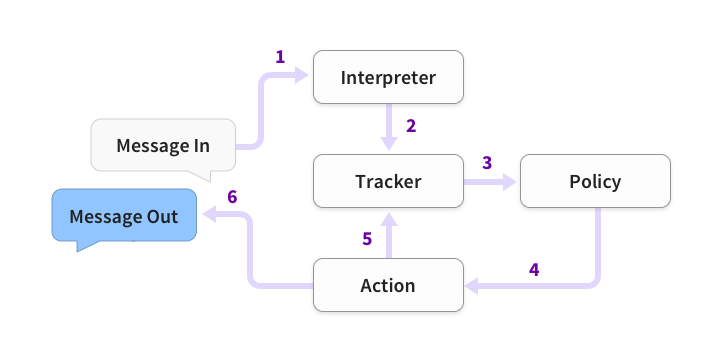
The whole backend had to be restructured to be adapted to the Rasa platform.

### 4.1.1.2 Frontend

For the final design the frontend only received some changes, the new design, as the original, still uses Telegram as the chat platform. One aspect that changed was the webhook, which for the revised version comes integrated with Rasa, so it doesn’t make any sense to develop it.

### 4.1.1.2 Backend

As everything is designed following some rules, the changes the backend can have is limited to what Rasa offers.



The steps this new backend follows is different.

1. When a message is received it goes to the Interpreter, which in the original design would be part of the context manager, in the interpreter a dictionary is created which contains the original text, entities and the intent.
2. An object linked to the state of the conversation called Tracker receives the info that a new message has arrived.
3. The policy receives the state of the conversation with the user, this information comes from the tracker.
4. Depending on the state the policy choses what action to take.
5. The tracker logs the action taken from the bot.
6. The response is sent back to the user.\cite{rasa-arch}

- User authentication

- Change of model

- Rasa

- Custom actions

## 4.1.2 Database

- Tables  
- Language

# 4.2 Resources Used

# 4.3 Implementation

# 4.4 Testing