Currently we are using the Django WSGI, to have a distributed application you would need to use ASGI (asynchronous server gateway interface). First we must choose a django ASGI server, we can use either Daphne or Uvicorn. Secondly, we must divide tasks that can be done concurrently. By separating each of our django applications, to handle multiple processes in a Daphne server to power Django Channels. By developing an application to work as a two way connection with an open connection on the server, it can allow for the developers to create different, and better applications. We could have the client send geolocation updates, so that we as developers can track their run, and even suggest healthy food options for them. How would this be possible? Since Django's daphne servers can handle a websocket protocol, it would allow for us to create a TCP connection with the server, by performing a three-way handshake. This way we have a steady byte stream of data from client to server and back to the client. By switching to an ASGI server, it can allow our applications to do more than our competitors and keep us in business. These ASGI servers allow for the use of *'async'* and *'await'* and thus while we wait for a network response, our django server can handle other issues.

The scalability of our postgresql server is allotted by choosing to use a Multi-Master PostgreSQL. Due to global laws, we would need to have our physical databases inside our most popular countries (Moldova, Argentia, and New Zealand). To distribute our program globally, we could set up an EC2 instance in AWS. Then ensure that we deploy these programs to data centers nearby our most popular countries. All we would then have to do is set up for the database, django server, S3 buckets, then make our migrations. After this initial set up we would be tasked with monitoring and updating the system. This software architecture is most relevant for Weight book because, will need to handle multiple writes every single day, as well as the most Real-Time data as possible. We the developers are in charge of handling conflict resolutions. Because of the possibility that you can have shared data in different servers, The developers/administrators would require data to be organized in a way to prevent the creation of unnecessary data, by ensuring that writes are done in distinct rows. One of the most important aspects is how we can load-share multiple concurrent writes to a database. With a multi-master setup we can handle our write requests to the geographically closest master server. If there is an overextended server, we can have it written to a backup server. It would then write to the other master databases concurrently. If the master node were to crash, the multiple copies of the master node would allow for incoming writes as the master node gets restarted to be used again. At the basis of keeping the company alive, a Multi-Master architecture needs to allow the user to be able to write or read from the database at any time. They may have inconsistent data on their end, but it will eventually be correct in the distributed system.

To help with distribution and everyone being able to run the project on their machine, we decided to use a docker container for our entire backend. This docker container allowed for us to easily start up the project, but it also can have more beneficial usages for a more distributed project. While we had decided to dockerize the entire backend, we can instead have multiple containers all running in the same docker. Each one of these containers would perform abstraction on a django application, allowing our applications: users, backend, auth and workouts to each run in their own container. This would be better for Weightbook, because containers can be much less resource intensive and allow for us to achieve horizontal scalability, with a cheaper cost in cloud services. By utilizing a docker swarm, these different applications can have shared resources when a client needs to login and then write in their workout. The docker swarm will handle most of the load balancing for us, but can also allow us to deploy more resources to certain servers and thus be market efficient.

In conclusion, to achieve a distributed system, you would need to perform 3 major changes to our code base. You would first need to change the interface of our Django server to be an ASGI, to handle asynchronous code. This not only helps our global constraints, but also allows efficient resource usage (a major cost for software companies). It also would benefit us by allowing us to create different and better products with a TCP connection instead of just http requests. The second code change would be to have a distributed database. When deciding how to handle our servers, because of global constraints to ensure data is served in a timely request, we would need to have a multimaster architecture. It allows for multiple nodes to be written to and read from, that can load share to other servers, when there is overflow or a master node goes down. Lastly, one of the last aspects to become a more distributed program would be to use docker containers for each of our applications. By using docker containers inside of each application, it can allow for the shared resources of the operating system and shim. These different containers can hold or share information between containers as needed. These different distribution aspects would not only allow for better performance, but also allow our developers to create better features that keep users using our platform.