



Faculty of Information Systems and Applied Computer Science

Chair of Mobile Systems

University of Bamberg

Master's Degree in Applied Computer Science

Master Thesis

A Simulator for Location-Based Agriculture Scenarios

Author:

Paul Pongratz

Matriculation Number:

2028489

Reviewers:

Prof. Dr. Daniela Nicklas

Simon Steuer, M.Sc.

Bamberg, March 29, 2023

Abstract

Digitalization is finding its way into every field of industry and business, including agriculture. Due to recent R&D efforts, many agricultural processes can be enhanced using digitalized technology. One of the possible advancements is smart localisation techniques for farm animals.

This thesis summarises the research and development carried out to create an initial prototype of a farm animal location simulator. This work is in the scope of the WeideInsight project, which is run by the Chair of Mobile Systems at the University of Bamberg, Germany cooperating with other universities and companies. The project focuses on finding energy-effective localisation techniques for locating cattle in barns and on pastures. One component in the project plan is a simulator, mainly to demonstrate the system functionality to potentially interested customers.

We target the creation of such a simulator fulfilling all requirements defined by the project plan. After an opening exploration of the simulation topic, we decide to use the software SmartSPEC as the simulation engine. We implement a backend consisting of microservices, which uses SmartSPEC internally to create simulations. For the frontend, we implement a Web UI using the framework React, which accesses the backend via a REST API.

The evaluation of our prototype shows that it can create realistic simulations for various scenarios, but also highlights limitations in the movement frequency, making it incapable of simulating fast-moving entities.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Statement	2
1.3	Methodology	3
1.4	Outline	3
2	Background	5
2.1	Simulation	5
2.2	Smart Spaces	12
2.3	Project WeideInsight	14
3	Requirement Elicitation	17
3.1	Methodology	17
3.2	Simulator Feature in Project Specification	18
3.3	Stakeholder Meeting	18
3.4	Finalized Requirements	21
4	Solution Architecture	24
4.1	Backend	24
4.2	Frontend	31
5	Implementation	39
5.1	Backend	39
5.2	Frontend	47
6	Evaluation	54
6.1	Experiment 1: Testing Different Settings	54
6.2	Experiment 2: Simulate Ground Truth Data	61
6.3	Requirements Fulfillment Check	64
7	Conclusion	66
7.1	Summary	66
7.2	Future Work	66
	References	68

A Sources	71
B Experiment 1 - Test Case 1: Settings	72

List of Figures

1	Thesis Methodology.	3
2	Entities and Relationships in the M&S Framework.	6
3	Architecture of a GAN.	10
4	Components of SmartSPEC.	14
5	Geovisualization of Bluetooth Beacons and Location Measurements in the WeideInsight Experiment.	16
6	WeideInsight Architecture Including the Simulator.	20
7	Simulator: Server-Side Architecture.	25
8	Workflow of the <i>GET /training/data</i> Endpoint.	29
9	Scenario-Learning Workflow.	30
10	Scenario-Generation Workflow.	31
11	Design-Sketch of Navigation Bar.	35
12	Design-Sketch of Execution Page.	36
13	Design-Sketch of Map View Page.	37
14	Design-Sketch of Settings Page.	38
15	Code-Structure of Location Service.	43
16	CSR Pattern.	44
17	Code-Structure of Frontend React Application.	50
18	Web UI Page: Settings.	51
19	Web UI Page: Execution.	52
20	Web UI Page: Map View.	53

List of Tables

1	Functional Requirements for the Simulator.	22
2	Non-Functional Requirements for the Simulator.	23
3	Test Cases for Experiment 1.	57
4	Results of Experiment 1.	60
5	Results of Experiment 2.	63

Abbreviations

API Application Programming Interface

CNN Convolutional Neural Network

CPFT Central-Place Foraging Trajectories

CSR Controller-Service-Repository

DRY Don't Repeat Yourself

EMA Exponential Moving Average

GPS Global Positioning System

HMM Hidden Markov Model

HMS Herd-Management System

HSMM Hidden Semi-Markov Model

HTTP Hypertext Transfer Protocol

I/O Input/Output

IoT Internet of Things

IPS Information Processing System

IT Information Technology

KM Kilometres

LPWAN Long Range Wide Area Network

LSTM Long Short-Term Memory

M&S Model & Simulation

MVC Model-View-Controller

OS Operating System

REST Representational State Transfer

SMA Simple Moving Average

SOAP Simple Object Access Protocol

STC Space Time Cube

UI User Interface

1 Introduction

It is omnipresent that our world is steadily digitalizing. This development started roughly 30 to 40 years ago and continues even today, making the 21st century often called the "Digital Era". From a technological standpoint, digitalization means being connected; with systems, humans or animals interacting with each other. Businesses thrive to connect their products to increase efficiency and revenue, not only in the Information Technology (IT) sector but also in areas where 30 years ago not many would have thought it would be possible, like health or agriculture. This innovation was largely empowered by the emergence of the Internet of Things (IoT) [1]. IoT is a general term used for devices connected to the internet but not actively operated by a human, so-called "Things" in this case. IoT helps interconnect more and more different systems and entities to redefine solutions which previously worked without the usage of the internet or connected devices.

After the term IoT was introduced by Kevin Ashton in 1999, the developed solutions implementing the new idea got more complex year by year [2]. As science and technology advance, the development process in the IoT field requires us to deal with more and more data and systems which interact with each other. Often simulators are used to ease this process by mimicking a real system during implementation of the actual logic. In many cases not a whole system has to be simulated, but only an incoming data stream which provides realistic but artificial data. For example, if we implement a fire alarm system, it would be inconvenient to be required to light a fire to test our application. Instead we provide fake sensor data mimicking a realistic fire breakout scenario [3]. The creation of realistic simulators for these purposes is a research topic in itself, which will be the focus of the upcoming content in this work.

1.1 Motivation

Digitalization is continuously improving nearly every section of the industry, which also includes agriculture. While this is a fairly new research area for digital applications, there are many use cases to replace manual effort with automated digital systems. In the field of dairy cattle farming recent studies tried to use IoT technology to analyze the behaviour of cattle. The old-fashioned way to determine the behaviour is to manually observe the cattle, which requires daily effort and professional knowledge. If farmers could get rid of this workload, more cattle can be kept in the stable and other tasks can be done instead of continuous surveillance. To get this work done by machines, we have to find out which information is needed to determine the behaviour of cattle. What it boils down to is, on the one hand, the location and on the other hand the movement of the animal over a certain

period of time. Researchers try to find ways to efficiently determine these measurements to then create new information out of it, which includes the behaviour among other things.

The Chair of Mobile Systems at the University of Bamberg conducted multiple studies to automate behaviour recognition of dairy cattle. In the first project the focus was to classify cattle movements to a specific action, e.g. grazing or lying, using machine learning algorithms [4]. Now in a second project called "WeideInsight" efficient ways to determine the location of cattle are researched [5]. This is of high importance because current ways to locate farm animals are mostly using high-power Global Positioning System (GPS) sensors or other methods which use a lot of energy. The problem with this is, that the sensors attached to the cattle need to be battery-powered. So, this project is searching for a method to determine cattle locations with a low energy footprint. The final platform should allow the farmers to observe the locations of the cattle via a map in an intuitive User Interface (UI).

Part of the final product should be a simulator to showcase the functionality of the system to potentially interested customers. Our work focuses on the design and implementation of this simulator according to the project's requirements.

1.2 Problem Statement

The WeideInsight project has the goal of creating a complete solution for farmers to track their cattle. For the users, the system is portrayed through a UI, which shows the movements of the cattle on a map. Before this can happen, the whole system including all sensors and other components has to be set up and configured on the farm. This was identified as a problem during the project because customers want to see how the system works before setting it up in their local environment for days. That is where a simulator comes into play, to provide potential customers with a first look at the system without installing the hardware. To advertise the benefits of the system it is also the intention to provide a realistic simulation, which takes the actual map of the customer's barn and their cattle into account. The simulator should give the user the imagination of the actual in-place system, but work with simulated data in the background. Otherwise, the simulator can also be used during development to implement higher-level components which need realistic data to be tested.

So, the goal of this thesis is to find out how the described simulator can be implemented and which technologies are needed for it. A working simulator would resolve the need for a complete setup of hardware and software at the customer's farm to see how the system works in the local environment. The simulator can then be used for simulating the tracking of any farm animals and should not be limited to the cattle use case. Taking these aspects into consideration, this thesis has the overall goal of answering the following question:

How to develop a simulator for location models used in modern agriculture?

1.3 Methodology

Before starting the development process of the simulator, we have to find out which aspects have to be considered to create a realistic simulation. This includes also research on which role the available space, e.g. the barn, plays in generating realistic data. The gained knowledge will be used to find an appropriate framework to generate new data, by judging the realisticness of the result. With a logical idea of the possibilities in place, the requirements will be elicited by taking the project goals and stakeholder demands into account. After that, the architecture of the simulator, which should fulfill the requirements, will be designed. This involves all relevant decisions for the implementation, e.g. which programming languages, tools and frameworks will be used. Then the system will be implemented in one iteration, with the goal of a working first prototype. The quality of this prototype will then be evaluated by executing two experiments judging the different settings the simulator can deal with and the realisticness of the returned locations. By collecting the evaluation results and missing points, the requirements for a second implementation iteration in the future will be given. Figure 1 shows a simplified view of the methodology of this thesis.

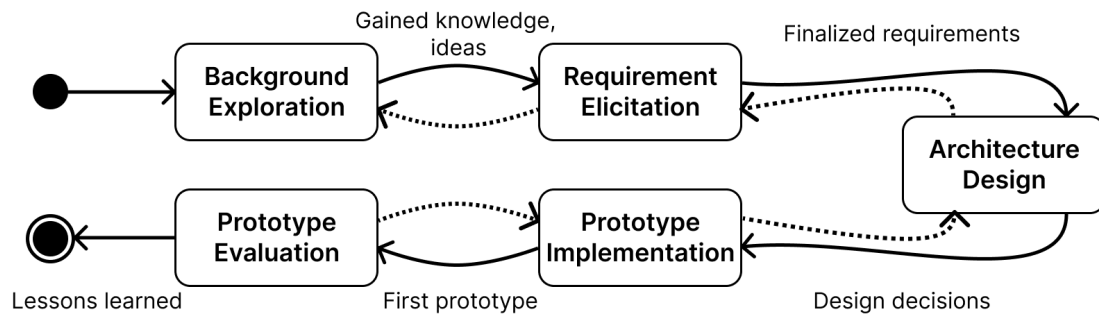


Figure 1: Thesis Methodology.

1.4 Outline

This section provides a summary of the upcoming chapters and their content.

In chapter 2 literature in the fields of simulators/simulations and also smart spaces will be examined to gain knowledge of how the simulator should be designed. It finishes with an evaluation of the software SmartSPEC, which will be chosen as the framework to generate new location data.

In chapter 3 the requirements for the simulator are collected from the project specification and a meeting with the stakeholders.

In chapter 4 the architecture of the simulator is presented for both the server (backend) and the client side (frontend). For the backend we focus more on the internal structure, for the frontend more on the design and data representation.

Chapter 5 is about the implementation of the simulator including design decisions and the structure of the code. This is also split into frontend and backend sections each including a guide how to use them to create simulations.

The implementation is then evaluated in chapter 6 by running two experiments. With the results of both it is checked if the defined requirements are fulfilled.

Chapter 7 concludes the thesis and hints towards future work which could be done especially regarding implementation improvements.

2 Background

In this chapter, we will explore literature helping us to understand how to design a *simulator*. In section 2.1 *simulations* in general will be discussed, first from a theoretical viewpoint and then their applications. Section 2.2 brings the focus to the type of data we want to simulate, which is a *smart space*. The idea is to abstract locations to the presence in *smart space*. The presence in a *smart space* can be simulated using the software SmartSPEC, which we evaluate to use it as a simulation engine in the simulator. In the upcoming content of this work the connection we establish between *simulations* and *smart spaces* is described further. Finally, the project WeideInsight, which this thesis is integrated in, is presented including the current status and planned features.

2.1 Simulation

To find out how a *simulator* works, it is helpful to first look at the product we want to create, a *simulation*. This section will discuss what *simulations* are, why we need them, and how they work.

2.1.1 Definition

Simulation has many facets and is hard to define in one line, as there are countless aspects involved in this matter, which also change over time and differentiate according to the field, in which it is applied in. Ören [6] tried to list a summary of all relevant definitions of the term *simulation*. It can be observed that the definition greatly depends in which field the authors worked in and at which time it was said. Early definitions from 1960 onwards state that a *simulation* is a representation of aspects in the real world by semantics or through a computer. From our view over 60 years later, this is a very context-sensitive definition and not applicable for the whole *simulation* field in our current time. Later around the year 1980, the definition shifted towards seeing *simulation* as the creation of *models* over time, where the source is not specified like it was previously with the real world. So in other words, *simulation* is seen as a continuous process which creates a representation of something "real". Defining it like this brings in the coherence with machine learning algorithms, which work in a similar way.

Let us now dive a bit deeper into the different entities involved in a simulation, like *model* and *simulator* and define how they interplay to create a *simulation*. Ören contains all of the involved entities in a Model & Simulation (M&S) framework and links them together with relationships (cf. figure 2).

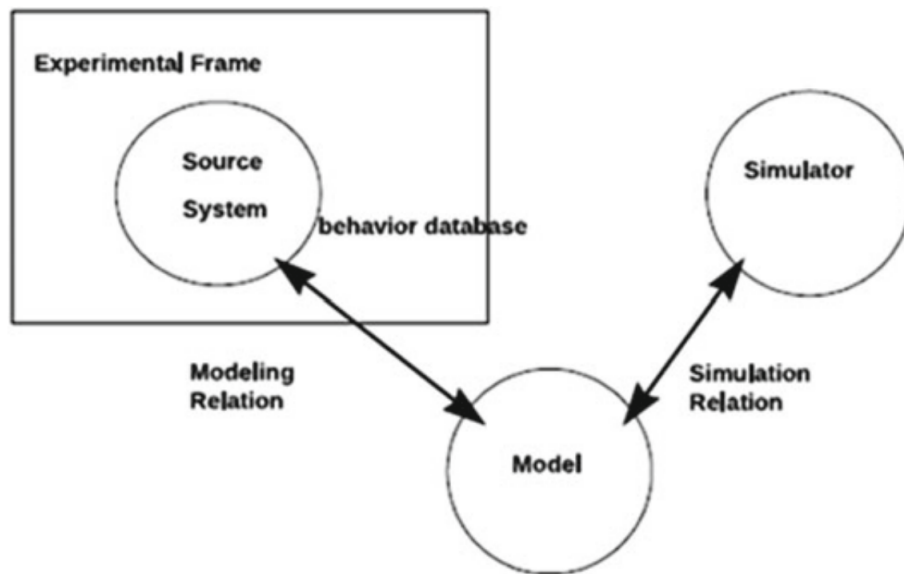


Figure 2: Entities and Relationships in the M&S Framework [7, p.6].

The different entities in the framework fulfill specific roles to be able to create a *simulation* [7, p.6-10]:

- **Source System**

The environment which we want to simulate. Another word for this is *simuland*. The other entities see it as a source of data, which can be observed.

- **Behavior Database**

All data that is emitted and observed from the *source system* makes up the *behavior database*. The data must be indexed by time and can be both real or artificial.

- **Experimental Frame**

Decides which data from the *source system* is kept in the *behavior database*. This happens based on conditions, which define the data attributes of interest for the *simulation*. Here the objectives of the *simulation* can be adjusted by specifying the relevant data in the conditions.

- **Model**

The set of instructions to generate simulated data out of the input data from the *behavior database*. Can be seen as Input/Output (I/O) logic, which does not contain software to execute it. It is a mathematical translation function which takes the input data as a parameter and returns simulated data. This translation function is commonly built using machine learning algorithms, which can even adapt over time to the input data.

- **Simulator**

The agent which executes the *model* and generates simulated data according to the *model's* instructions. Generally, this agent is a software application.

All of the entities interact to solve the task of creating a *simulation*. The *simulator* executes the *model* which generates simulated data based on the original data in the *behavior database*. The data originates from the *source system* and is filtered by the conditions specified in the *experimental frame*.

2.1.2 Why Do We Need Simulations?

Simulations are used for various things, which can be distinguished by the goal we are trying to achieve. Ören lists three different types [7, p.2]:

- **Running experiments**

The most common use of *simulations*. The experiments meant in this scenario are run on the *models*. In easy words, they answer the question "What would happen?" without trying it out in the real world. Using *simulations* to answer this question helps find out the limits of the *simuland*.

- **Gain experience**

Simulations can be used to gain/enhance different skills, e.g. decision making, communication and operations. A good example is a flight simulator to train pilots to fly a real airplane.

- **Imitation**

When we try to make the viewer believe the real system is being executed, but they just see a *simulation* of it.

The author is not explicitly listing demonstration, which is our main goal. But it can be seen under the **Gain experience** goal, as it also provides the viewer with new information to learn about.

Aside from the goal which a *simulation* should achieve, there is the question of why it is even necessary to run a *simulation* instead of conducting an experiment in the real world. This is mostly related to the data which we simulate being too difficult to gather, which makes a *simulation* the easiest solution. Let us explore some common situations, where real data is a struggle or infeasible to get:

- **Setup cost**

The cost of setting up all required hard- and software for a system is often too high to

do it just for testing purposes or to "try it out". This is especially the case for pervasive computing applications, which generally involve many hardware components like sensors, actuators and gateways. There are so many parameters which have to be accounted for, which would result in countless different test cases required to run. Doing this in an automated way using a *simulator* is mostly the best or only solution [8, p.1-2].

- **Staff cost**

Especially in long-running tasks the staff costs can be very high. Running a *simulation* instead, which also can be fast-forwarded, could reduce costs and improve the workload.

- **Data privacy**

Collecting data in the public space always involves us handling data privacy according to the local law. Especially when location data, like mobility trajectories of humans, are involved, a *simulation* might be needed during development, because of the risk of privacy breaches [9, p.1].

- **Ethical issues**

A *simulation* can help detect and prevent running into ethical problems while working in a controversial field.

2.1.3 Algorithms

The *model* in an M&S framework contains the logic of how new data is generated with the information given from the *behavior database*. To accomplish this, an algorithm is needed to create realistic data. The choice of which algorithm to use mainly depends on the system which we want to simulate and there are none which would fit every use case. So, to explore this topic, we will focus on the type of data we are dealing with: Location models. In this section, we will explore the state-of-the-art algorithms, which are currently used the most for location/trajectory generation. Most of them were extensively studied to simulate human mobility, but there are also papers available exploring their functionality for animal mobility cases.

Hidden Markov Models (HMMs)

An HMM is a statistical model, which uses a Markov process with hidden states. An input is transformed by taking on multiple states till an output is produced. HMMs have been proven to produce good results in simulations of Central-Place Foraging Trajectories (CPFTs). CPFTs are characterised by the individual coming back to a central location regularly, which produces trajectory loops. Sadly in larger spatio-temporal scales HMMs struggle, because they are only calibrated at a local scale, meaning a fixed area [10]. Another

downside is that HMMs only operate with the knowledge of statistics and cannot take individual behavior into account [3, p.2]. This means, that every entity we simulate, in our case farm animals, would have the same behavior and move around in a similar fashion.

Generative Adversarial Networks (GANs)

To model behavior of individuals, a more "smart" logic is needed than what state-driven approaches like HMMs could provide. Recent simulation literature shifts towards preferring data-driven approaches with machine-learning techniques. Especially deep learning makes it possible to accurately represent complex data features in a model. Deep-Learning makes use of neural networks with multi-layer processing units, which constantly adapt to the data, which is flowing through them.

So-called "Generative Adversarial Networks" (GANs) are favoured to be the next big thing for data generation [10, p.2-3]. GANs consist of two neural networks, which are trained separately: One is responsible for generating new data, called the *generator*, while the other serves as a decision maker, if incoming data is real or artificial, called the *discriminator* [3, p.2]. The model is trained as long until the *discriminator* gets "fooled" more than 50% of the time that an artificial trajectory is real. This means that the *generator* is trained well enough to produce realistic data, which the *discriminator* cannot distinguish from the real data most of the time [11]. GANs are compatible with a vast amount of deep learning algorithms, including the two most popular ones Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN). The algorithm can be freely selected for both *generator* and *discriminator*. The architecture of a GAN is quite simplistic as displayed in figure 3.

Even though GAN is probably the best simulation method in terms of realism, Chio et al. argue that there are some significant downsides. They state, that the semantics regarding the space, which we want to simulate, and the "events" the artificial individuals participate in are not enforced [3, p.2]. By events, they mean specific actions the simulated entities take part in. This is a crucial factor for simulating herd behaviour. They highlight a valid point because deep learning is kind of a black box algorithm; meaning it is unclear how it internally works because of its complex logic which is ever-changing. Another downside is that GANs are very difficult to set up and train and there are methods which need less effort.

Semantic models

Instead of relying on machine learning models to build our model (GAN) or only creating a statistical model with HMM, we can define the semantics which makes up the model ourselves. In this way, we create our own logic for how new data is generated. This gives us a lot of flexibility because adjustments are easy to implement. This method is useful when we want to enforce specific semantics, e.g. space constraints, in the model.

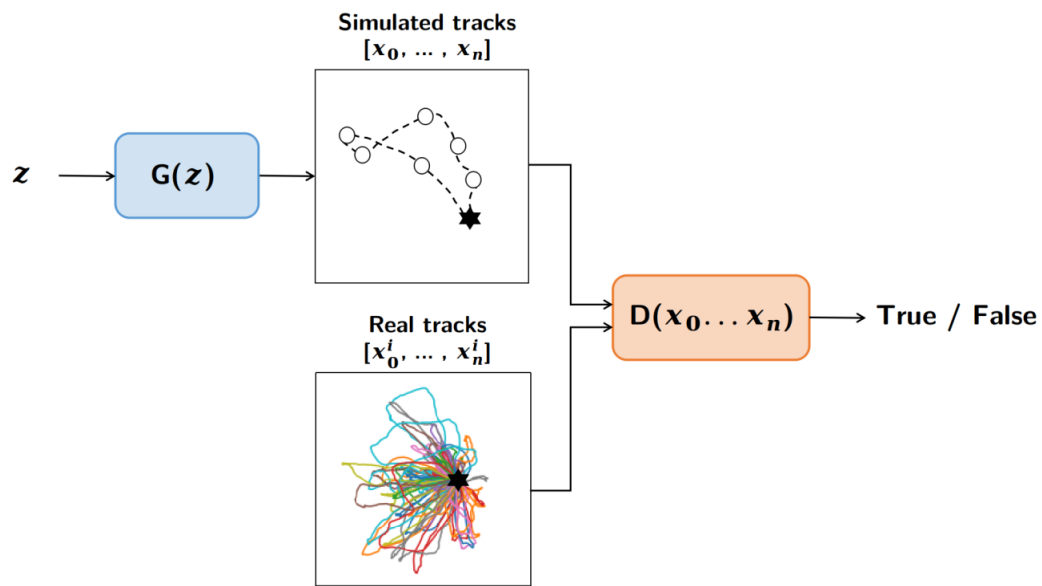


Figure 3: Architecture of a GAN [10, p.3].

The *generator* G takes a random noise vector z as input and outputs the generated data x_0, \dots, x_n .

The *discriminator* D then decides if x_0, \dots, x_n are artificial or not.

2.1.4 Examples

Let us explore how the simulation concepts are used in different examples. The focus will be on location models as it is the theme of the thesis.

Bird Trajectory Simulation using HMM and GAN [10]

This work by Roy et al. runs simulations for location data of tropical booby seabirds. The data was collected during their breeding period and contains mostly foraging trips on the sea, where the birds search for prey and then come back to the colony. This means the trajectories produced by the birds are CPFTs. The author compares the simulations of this data when using an HMM or a GAN as the framework to create the model. In general, the GAN could produce more realistic data. This judgement was made based on the fact that the mean spectral error was lower in comparison to the HMM. HMM only outperformed GAN in the accuracy of local-scale descriptive statistics, e.g. the step speed. The result of this study aligns well with the definitions from subsection 2.1.3, as the GAN was proven to produce more realistic simulations and HMM to do well in descriptive statistics.

Human Mobility Simulation [12]

This paper proposes a human mobility modelling framework called "DITRAS", which simulates the mobility of an individual taking spatio-temporal patterns into account. It works in

two phases: The first phase creates a "mobility diary", meaning it only generates temporal information. The second phase then creates new trajectories with the diary and the available space as inputs. For the first phase, an HMM is used, which describes the probability that an individual follows their routine or visits an abnormal location. After that, the second phase uses spatial tessellation to generate locations. Spatial tessellation defines a relevancy for each point in the defined space. If the relevancy is higher, it is more likely that the individual will move to that spot. The study resulted in a good simulation of human mobility but lacks behavior heterogeneity across the simulated individuals. The applied method is also very complex because the spatial tessellation relevancies have to be defined for the whole space.

Visual Tracker based on Hidden Semi-Markov Model (HSMM) [13]

In this work, the authors Ha and Kwon improved a visual tracking system with predictive trajectory simulation. Based on the latest real trajectories, the following trajectories are simulated. They are then used as a focal point for the tracker instead of the real trajectories, which greatly enhances the smoothness and accuracy of the tracker. This was claimed, because the real trajectories contain a lot of noise, meaning that the trajectory is quite devious, while the simulated trajectories are straighter, indicating the approximate direction. Instead of an HMM, they used an HSMM, which additionally allows multiple observations in one state, while the HMM does not. This tweak greatly increased the accuracy of the simulation.

2.2 Smart Spaces

Apart from the topic of simulation itself, we need to look into the thing we want to simulate, which is a smart space. After exploring the definitions, we will survey their application in research. Finally, we will evaluate the smart space data generator SmartSPEC by Chio et al. for its applicability to our animal location simulation use case.

2.2.1 Definition

Smart Space is a term which came up during the development of advanced IoT solutions. While there is no common definition, authors mostly identify a smart space by its characteristic of being monitored by sensors and the ability to react to occurrences in the space, e.g. people entering the space, the temperature reaching a defined threshold, etc. Chio et al. give a specification by defining entities in a smart space ecosystem [3, p.1]:

- **Space**
The geographical layout of the smart space.
- **People**
The people inhabiting the smart space.
- **Events**
The actions of people in the smart space.
- **Sensors**
The observers in the smart space.

This is a definition which is of course tailored for the use case of observing people and their actions in defined areas. Nevertheless, we can standardise this definition by replacing people with entities and actions with occurrences. This makes it coherent with our use case of smart monitoring of animals.

There are also other views about smart spaces, focusing on the user, which are the people in Chio et al.'s definition, instead of the space. Balandin and Waris define a smart space by the interaction a user does with its surrounding environment [14]. The environment offers services, which respond in various forms according to the user's needs. The user can access these services to enhance the experience in a space.

We can extract out of the space and user-focused definitions, that a smart space is a highly technologized geographical space, which deals with information about the entities contained in it. The definitions tightly couple the space and the entities which inhabit it.

2.2.2 Application in Research

Not every paper uses the word smart space to describe the environment of our definition, but there exists a lot of research material dealing with the problem of how entities and the space around them can correlate using technology. This is not a new topic and has been researched for centuries now. It finds application in smart cities and smart homes, for example. While there has been extensive research on this topic for centuries Helal and Tarkoma criticise that there are still very few smart cities and homes in comparison to the potential they have [15].

2.2.3 SmartSPEC

SmartSPEC is a smart space simulator and data generator. It is proposed in a paper by Chio et al. [3] and the source code is available on Github [16]. It is designed to simulate *people* participating in *events* in a defined list of *spaces* surveilled by *sensors*. Instead of relying on an HMM or machine learning algorithm to train the model correctly, they use semantic models, which are self-defined algorithms. This way constraints in the simulation can be ensured because the algorithm is easy to understand and customize. Using, for example, a GAN makes it hard to keep rules for the simulation, as a lot of parameters are generated randomly.

The functionality of SmartSPEC is split into two components: *Scenario-Learning* and *Scenario-Generation*, with *Scenario Learning* being the first step. *Scenario-Learning* takes ground truth data as input alongside a definition of the *spaces* and other settings to specify how an *event* in the data looks like (e.g. how long does an entity need to stay in a space that is counted as an event). The execution of *Scenario-Learning* generates internal data which is needed for *Scenario Generation* later on. This includes mainly *events* and *people*. The algorithm aims to resemble the ground truth data accurately in the generated internal data. The authors also mention the option to provide the internal data ourselves and skip *Scenario-Learning*. This is especially useful if we have no ground truth data and only a *spaces* configuration. With the internal data in place, we can start the second component, *Scenario-Generation*. This consists of multiple separate generators, an *entity generator*, a *synthetic data generator*, a *trajectory generator* and a *sensor observation generator*. The internal data is fed to all of them to solve the overall task of generating new location data. But the output does not include the coordinates, only the presence in a space. Figure 4 contains a component model visualizing this process.

As this software provides huge potential to simulate locations, we evaluated if it is applicable in our use case. The doubts, which of course arise, are, that SmartSPEC is designed to simulate humans and not farm animals. To find out how SmartSPEC performs with animal trajectory data, we provided the ground truth data of a previous WeideInsight experiment to

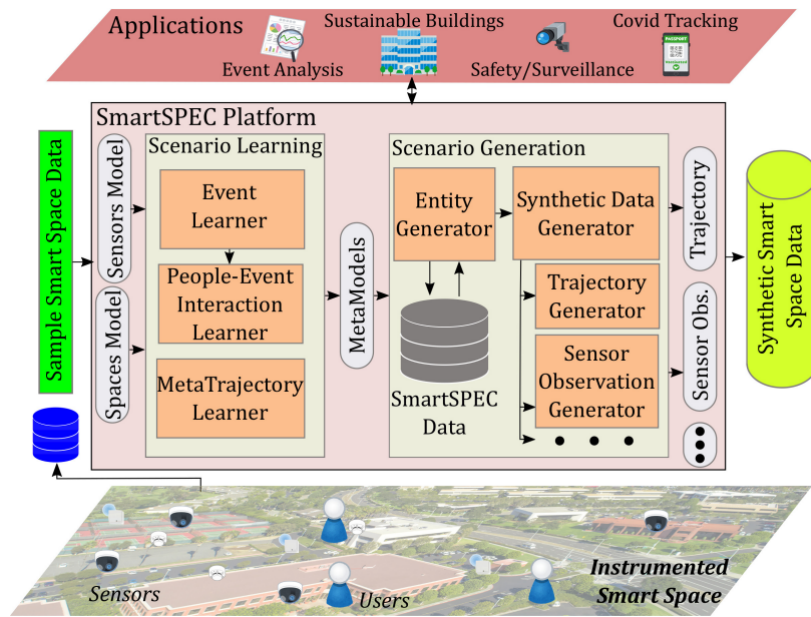


Figure 4: Components of SmartSPEC [3].

Scenario-Learning. After it successfully generated the internal data, *Scenario Generation* returned new location data as well. At the first glance and by trying out different parameters this data seemed to be promising. As we did not have a way to evaluate the data properly yet, we decided to go ahead and use SmartSPEC as the simulation engine for our simulator software. Later experiments in chapter 6 have to prove that our first impression was correct.

2.3 Project WeideInsight

This section presents the planned features and progress of the WeideInsight project. Our work is in the scope of this project to simulate its functionality to customers and developers.

2.3.1 Project Overview

WeideInsight is run by a consortium of multiple universities, small to medium-sized companies and state agencies. All of them are involved in the research and development of the project to solve the problem through a common interest. Advancements in the project's topic of energy-efficient location methods would benefit every partner.

The project has the goal of adding value to modern agriculture with the integration of low-cost and energy-efficient localisation techniques for barn and pasture surveillance of cattle in comparison to using GPS [17]. Localisation of cattle can bring multiple benefits

to an agriculture business; from simple detection if every cattle was milked to long-term analysis of movement patterns for health management. There are already existing solutions, but they rely on high-power GPS sensors, which are battery-powered and attached to the cow's necklace. This is of course not a long-lasting solution because the sensor's battery will run out of energy in the space of a few days. WeideInsight wants to find more energy-efficient ways of detecting cattle locations. It was found, that for the two different environments, barn and pasture, not the same localisation techniques are ideal.

Localisation in the Barn

For locating cattle in the barn, the project members decided to implement a solution using Bluetooth. First of all, there are Bluetooth sensors installed at various points in the barn to capture signals from every possible point. Second of all, the cattle get Bluetooth beacons installed on their necklaces, which periodically send out Bluetooth signals. These signals are captured by the sensors and used to determine the location of the cattle. The strength of the signals indicates how far a cow is away from a Bluetooth sensor. By means of triangulation, the location of the cow can be calculated taking the three closest beacons as subjects [5]. The first studies showed good accuracy after normalizing the measurements.

Localisation on the Pasture

On the pasture, it is infeasible to use Bluetooth for locating the cattle, because the distances are far greater in outside areas than in the barn. Here it was decided to use a Long Range Wide Area Network (LPWAN) solution which can handle data transmission in wide areas easily. The main downside of LPWAN is the low bandwidth, but that is not a problem in this scenario where only minor data is transmitted and more or less the signal itself is of importance. For the implementation, they chose the LPWAN Mioty, which is quite new and considered modern [18]. Also on the pasture triangulation is used to determine the location, but with the Mioty signals as the subjects.

Experiment

After implementing the triangulation localisation techniques they conducted an experiment on a farm to try out the first prototype of the system. This was done in the summer season to be able to also collect data on the pasture, not only the barn. The required hardware, e.g. the Bluetooth sensors, necklace beacons and Mioty gateways, was set up on a selected farm. Then, data was collected over a time of multiple days. After evaluating the results it was apparent, that the barn data was quite accurate, while the pasture data did not meet the requirements just yet. In figure 5 we can see a visualization of the collected data in the internal tracking software "Track" of the responsible stakeholder. Later steps in the project,

which are currently in progress or planned for the near future, are influenced by the outcome of this experiment.

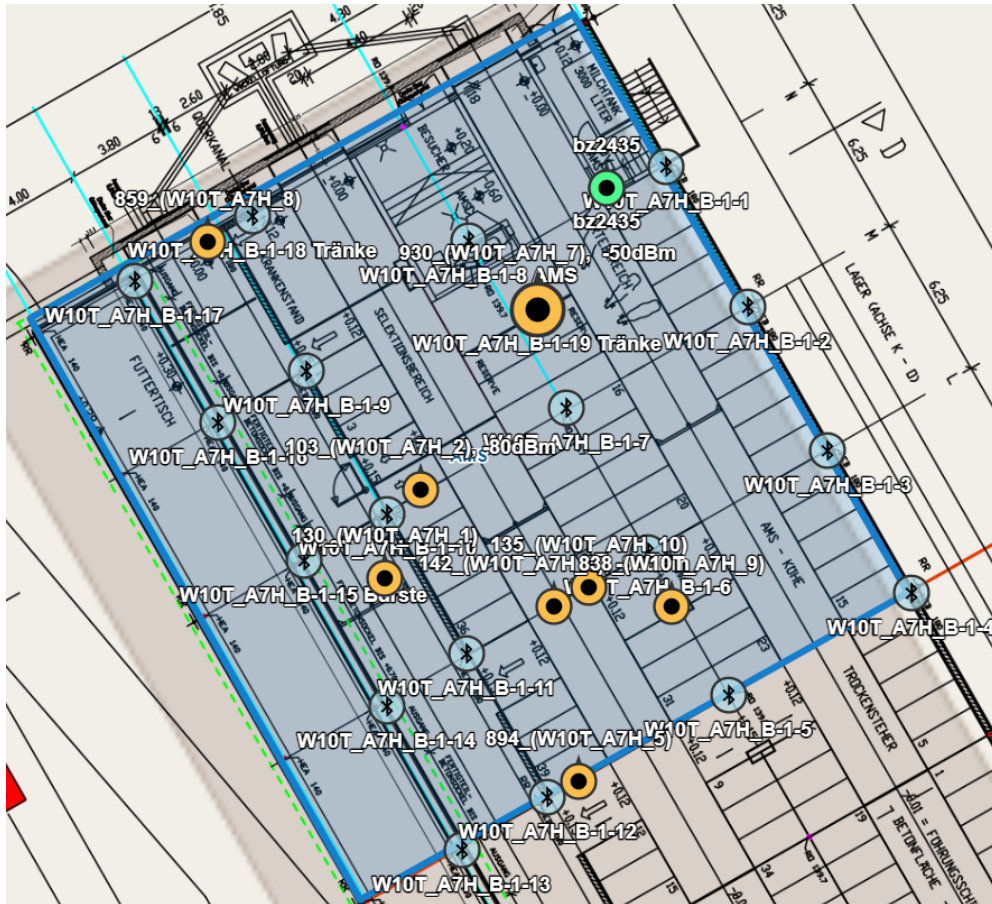


Figure 5: Geovisualization of Bluetooth Beacons and Location Measurements in the WeideInsight Experiment [19].

User Interface

The system should be visualized and maintained via a UI on the web or in an app. It should show the map in a similar fashion as portrayed in figure 5 and have all relevant settings available to configure the system.

Part of the project specification is also a simulator feature which will be discussed separately in chapter 3 as it is the part of the project scope which defines the requirements for this thesis.

3 Requirement Elicitation

Before thinking about how to design the simulator, the requirements for it have to be collected. As this thesis is in the scope of the WeideInsight project, the requirements need to align with the project goals and stakeholder demands. First, we define the methodology for the elicitation which will then serve as a guideline. Then the points regarding the simulator will be extracted from the project specification, which contains most of the requirements. In a meeting with the involved stakeholders, an initial idea will be presented, which is then refined based on the feedback. Taking all of the gained information into account, we can list the requirements for the simulator in the end. The fulfillment of the finalized requirements list will be subject to the evaluation in chapter 6.

3.1 Methodology

In this section, we define how we extract requirements from the project documents and meetings.

Types of Requirements

The goal is to have a list of functional and non-functional requirements. Functional requirements define the features of a product; so what a product must be able to do. In typical project planning workflows, they are captured with user stories. Non-functional requirements deal with the system, but not the actual product functionality. They are quality attributes defining the environment of the product.

Sources

As a source for potential requirements serves an extensive project specification with the goals, which were defined before the project started. In a meeting with the stakeholders, more ideas will be collected based on their feedback and the role of the simulator in the overall system will be discussed.

Generalization

Another goal of the work done in this thesis is to have a generalized solution that can be applied in other use cases in modern agriculture than cattle observation as well. The requirements should be generalized and not contain cattle-specific points.

3.2 Simulator Feature in Project Specification

The project specification lists the simulator as a separate feature and defines goals for it: It should be able to serve as a tool to trial, test, configure and demonstrate the technology in barn and pasture constellations [17].

The simulator should especially help medium or small farming businesses to check if the WeideInsight technology would be useful for them before implementing it on their farm. But it is also thought to be of use for developers of higher-level features to test and evaluate the features with different types of data. It should also be possible to configure the farm map of potentially interested customers as the base of the simulation. Furthermore, there should be settings available to adjust the number of simulated cattle and sensors. What should be simulated are realistic herd behavior and attributes of the location system, e.g. transmission distances, localization accuracy, and battery lifetime. The simulation should also be based on the currently used location system. So, the simulation should behave differently for the barn scenario with Bluetooth beacons and the pasture scenario with Mioty, also when they are used in combination or when GPS is used. In section 3.4 these points will be transformed into requirements.

3.3 Stakeholder Meeting

In the phase of collecting the requirements for the simulator, we held a meeting with all relevant stakeholders to discuss how the simulator should be designed and integrated into the system infrastructure. In a short presentation, we showed the ideas for a first implementation and asked the stakeholders to review them and give additional points which have to be considered. The thesis title caused slight confusion in the audience, as they expected it to only handle the cattle use case. This could be cleared up after clarification, that the simulator can be used generically for other agriculture use cases as well. We proposed to use Smart-SPEC as the framework for location generation and showed its basic functionality. It was highlighted that it simulates the presence in spaces, not precise locations. The stakeholders thought the tool to be promising but the simulation quality has to be verified in later tests. It was decided to go ahead with the original idea.

Another point in the discussions was how the simulator can be integrated into the infrastructure of WeideInsight. In previous iterations, the stakeholders already came up with a system-level architecture (cf. figure 6). This had to be adjusted now to include the simulator. The requirement is that the upper layers of the architecture, meaning the *Information Processing System (IPS)* and *Herd-Management System (HMS)* do not have to distinguish between real and artificial data.

To fit this requirement it has to be ensured that the data reaching the *IMS* has the same format as the real data published by *Track*. We evaluated that the best solution is to have a separate component, which transforms the data from the *simulator* in the same way *Track* does from the sensors, to make it compatible with the *IPS*. We named it *SimTrack*. Another option would have been to just send the data in the correct format to the *IPS* without an intermediate component. This would be less flexible than the chosen solution because the attached simulator is responsible for the formatting. If we want to attach a different, in the worst case third-party, simulator the formatting cannot be built into it. Then we would again rely on an intermediate component.

The simulator should have the option to create a simulation with real data as input. This data will come from *Track* and should be fetched from their Application Programming Interface (API).

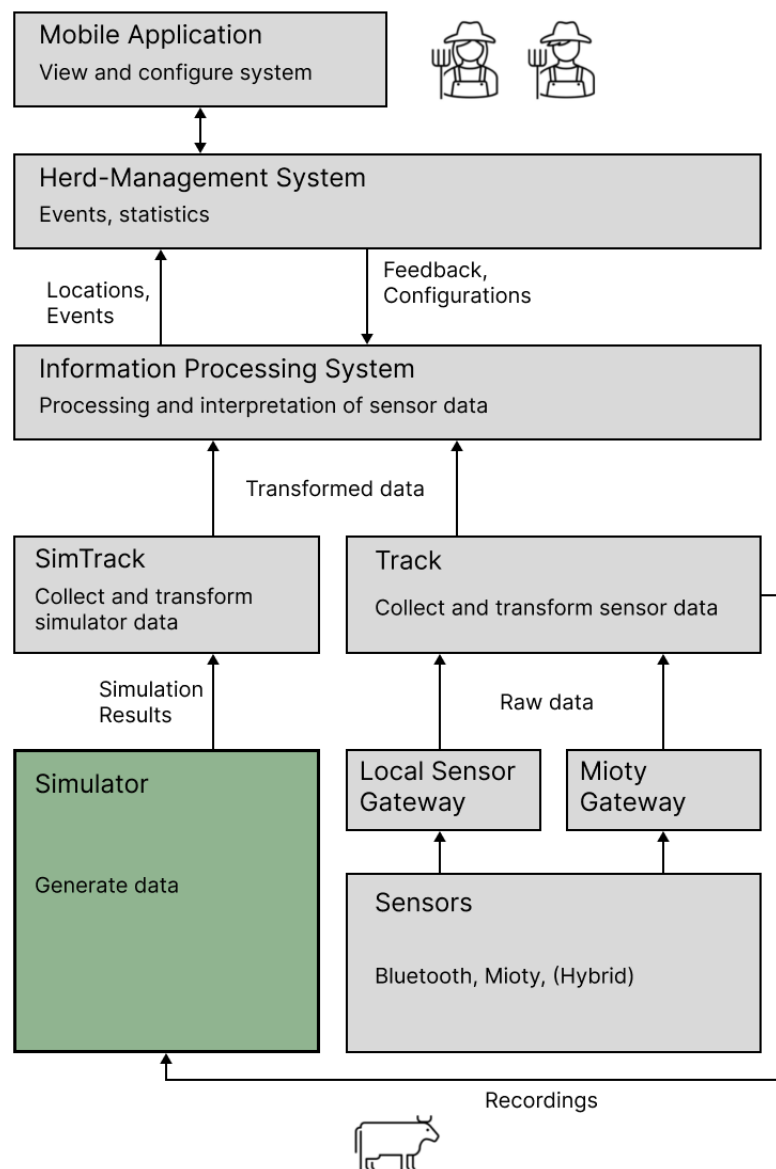


Figure 6: WeideInsight Architecture Including the Simulator.

Simplified component diagram of WeideInsight system. Sensors, which are either Bluetooth, Mioty, or hybrid senders, send measurements. Mioty data will be forwarded by a *Mioty gateway* to *Track* and all other data by a *sensor gateway*. *Track* collects the data from the different sources and transforms it into a universal format. The *simulator* generates data, which is transformed by *SimTrack*. From *Track* and *SimTrack* the data is sent on to an *IPS* which processes and interprets it to detect defined events. The events and location data are forwarded to an *HMS* which can be used, for example, to read out statistics or send notifications. The *mobile application*, which is the interface for a user, pulls data from the *HMS* and displays it on a map. It can also be used to configure parameters in the *HMS* and *IPS* to adjust data processing and eventing.

3.4 Finalized Requirements

The requirements for the simulator will reflect a summary of the previous sections. While the project specification already described many necessary features in theory, the meeting with the stakeholders revealed more technical points.

There are many ways to list requirements in different forms, for example, semi-formal models and natural language sentences. Which form to choose generally depends on the type of system we want to create [20]. Two of the most used methods are user stories and use cases. User stories describe what a user wants to do with software and use cases define the interaction between the user and the system. For our simulator, we decided to write user stories to define the requirements. We differentiate between two types of users, a general *user*, who is, for example, a customer simulating the WeideInsight application for his barn, and a *developer*, who is part of the WeideInsight team and uses the simulator for development and demonstration.

3.4.1 Functional Requirements

Functional requirements represent the features and capabilities of the system, meaning everything the system must do [21, Chap.1]. Our elicitation in the project specification and the stakeholder meeting revealed only functional requirements, which are listed in table 1. We assigned a priority to each requirement based on a subjective assumption of what is most needed. For our initial prototype, we will focus on fulfilling the requirements with priority *high* or *medium* especially.

3.4.2 Non-Functional Requirements

Non-functional requirements do not specify what a system must do but the way it should do it [21]. These are in general operational instead of behavioural attributes a system has. Areas of application are, for example, security or high availability.

The project specification and the stakeholder meeting focused on the functional requirements and did not specify details about non-functional requirements. So, we can apply here some general good practices for software products. We identified the requirements listed in table 2 to be needed for the simulator.

ID	Description	Priority	Source
REQ1	As a user, I want to simulate location models in barn and pasture scenarios.	high	Project Specification
REQ2	As a user, I want to configure the barn/pasture layout including spaces and sensors for the simulation and visualization.	high	Project Specification
REQ3	As a user, I want to adjust the number of simulated sensors and animals.	high	Project Specification
REQ4	As a user, I want to simulate realistic herd behaviour.	high	Project Specification
REQ5	As a developer, I want to easily integrate the simulator into the WeideInsight infrastructure. It should operate on the sensor level and the upper layers in the architecture cannot distinguish real and artificial data.	high	Stakeholder Meeting
REQ6	As a developer, I want to use the simulator to demonstrate the WeideInsight application.	high	Project Specification
REQ7	As a user, I want to use a straightforward UI to configure and run the simulator.	high	Stakeholder Meeting
REQ8	As a user, I want to use historic data as input for the simulation. This data should be fetched from Track.	medium	Stakeholder Meeting
REQ9	As a user, I want to simulate the used location system (Bluetooth, Mioty, and GPS).	low	Project Specification
REQ10	As a user, I want to simulate attributes of the used location system (e.g. transmission distances, localization accuracy, battery lifetime).	low	Project Specification
REQ11	As a developer, I want to use the simulator to test the WeideInsight application.	low	Project Specification
REQ12	As a user, I want to use the simulator to configure the WeideInsight application.	low	Project Specification

Table 1: Functional Requirements for the Simulator.

ID	Description	Priority
REQ13	The simulator's functionality can be extended easily.	high
REQ14	The simulator should be easy to install and run.	high
REQ15	The use of the simulator needs to be secure for the user.	medium

Table 2: Non-Functional Requirements for the Simulator.

4 Solution Architecture

With the requirements being set in chapter 3 we can define an architecture, which fulfills them. This is one of the crucial steps when creating new software. In many projects, there are often things implemented, which are not usable in the end, because they are not compatible with the rest of the system or added to the wrong component. A predefined architecture, which is strictly followed in the implementation, prevents later confusions like this and ensures a flexible and easily extendable software product as an outcome. This does not mean the architecture cannot be adjusted after defining it initially, but that all implemented features/components are represented in the architecture. This ensures that the architecture is an accurate view of the functionality of a product. This chapter contains the architecture of the simulator's backend and frontend. First, the broad architecture will be presented for both parts. After that follow the responsibilities of each component and how they link to other components.

4.1 Backend

The backend is the server-side software of the simulator and is responsible for doing all the logic to create realistic simulations. We will first discuss the architecture from a top level, e.g. where the backend is hosted, how it is accessed and which internal model is used to separate the logic. After that, we will discuss each component in detail and the endpoints they offer.

4.1.1 Architecture Overview

Figure 7 contains a component model of the server-side architecture, which we will examine step-by-step in this section.

How Do We Access the Simulator?

First of all, we have to answer the question of how the simulator's functionality should be accessed from the outside. This involves us thinking about which systems/users need to interact with the backend. In general, we could identify three different entities: The frontend, SimTrack and human users. The frontend should serve as the main access point for human users to use the backend. On the other hand, the combination of the backend and SimTrack should provide data for the IPS in the same format as Track does, although we only focus on the backend implementation, which serves the data for the proposed SimTrack component. As a third entity, users should also be able to run the simulator using the backend only to, for

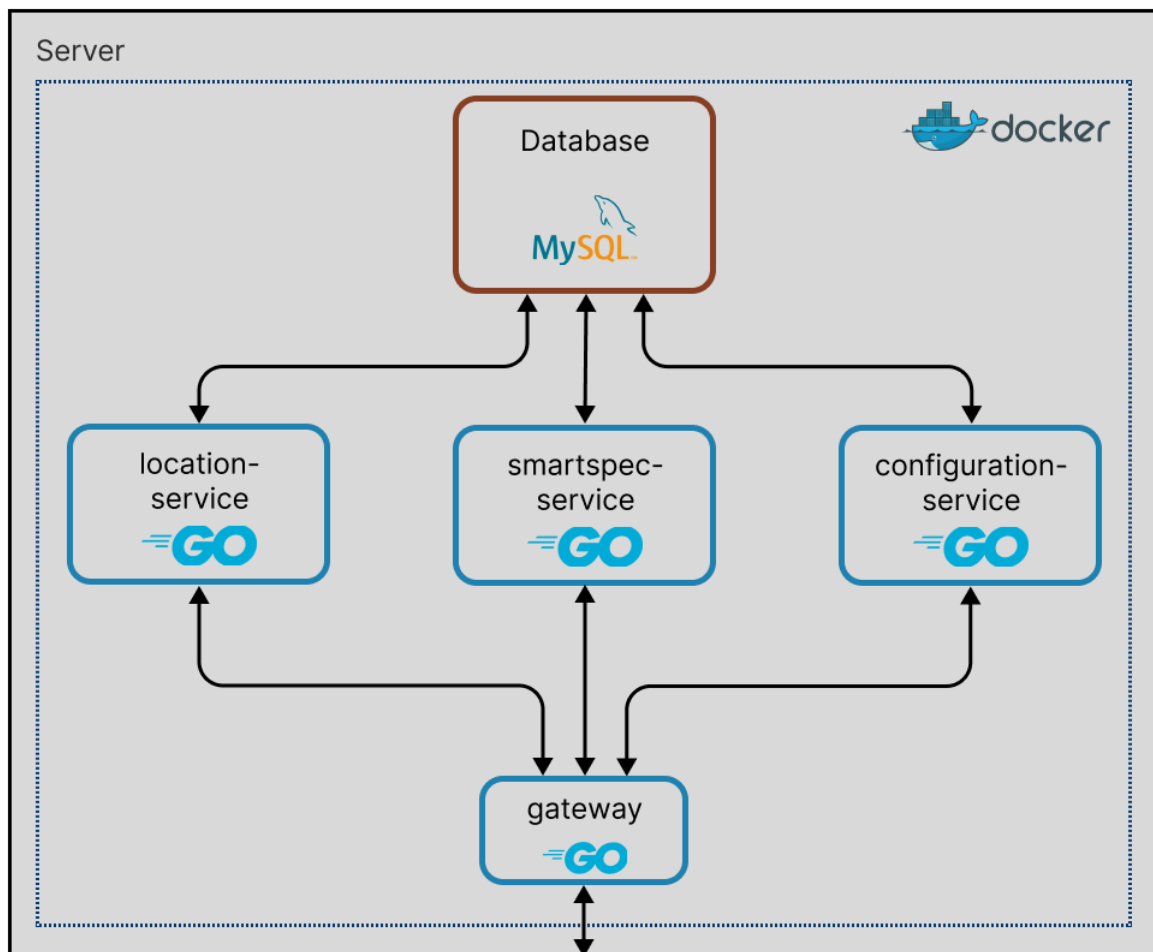


Figure 7: Simulator: Server-Side Architecture.

example, connect it to different systems. This emphasizes, that the backend should be usable in combination with and without the frontend. While SimTrack would access the data to use it in the WeideInsight system, we also want to provide an easy way for humans to use the backend without the frontend. There are multiple ways to achieve this:

- **Simulator as a Software**

The simulator runs as a separate software on the host of the system/user, which wants to access it. This means that we run separate simulators on each user's device.

- **Simulator as a Package**

The simulator is provided as a package to use in any compatible software. This involves every user/system importing the package into their software. For example, the frontend imports the simulator package and calls its functions.

- **Simulator as a Service**

The simulator is executed as separate software, which can be accessed through a common interface locally or remotely.

Out of these options, we identified **Simulator as a Service** as the best solution, because it decouples the backend fully from the user's host system. This is an advantage to Simulator as a Software, because it does not have to run on the same system where it is used, but can also be accessed the same way through the web. This provides the flexibility to run the simulator either in a central remote place or on the user's system. For our initial prototype, we will run the backend on the same host as the frontend, but keep the option to host the backend remotely for the future. To emulate running the backend on a remote host we will use Docker as an environment for all components of the backend. Apart from that Docker offers many other advantages which will be addressed in chapter 5 in more detail.

To make the simulator accessible as a service, it needs to provide an API. The most common paradigms for how to design an API are REST (Representational State Transfer) and SOAP (Simple Object Access Protocol). For our use case, we chose a RESTful interface, because SOAP does not decouple the logic as well from the frontend [22].

Using Microservices as Components [23]

Apart from the decision on how to access the simulator, we also have to decide how its internal model should look like. Here we decided to use a microservice architecture. This architecture consists of multiple loosely coupled programs, so-called microservices, which have different roles and solve the task by interacting with each other. Here lies the difference to a monolith architecture, which combines all the logic in a single program. A monolith is mostly distributed into components as well, but they all run in the same program. In a

microservice architecture, every component is run in its own service, making it possible to scale a service out, meaning to create multiple replicas of the same service during runtime. To achieve this in a monolith we would at least have to restart the whole system once. That is why microservices became the de facto standard for any bigger-scale software products.

In our microservice architecture, there are logical services, which provide the endpoints, and non-logical services, which are in our prototype only the database. The logical services are implemented using the programming language Go (cf. chapter 5 for details) [24].

4.1.2 Internal Services

Now we will look into the internal services of the simulator backend and discuss the role of each of them.

Database

To persist all data used in the simulator, we need a database. As the data is quite diverse, ranging from simple configurations to sensor data, a database which can handle these different structures is needed. The decision was made for the object-relational database MySQL, which can fulfill this requirement and is easy to integrate with the other components.

The database will contain the following tables:

- **measurement**

Contains the ground truth data collected in the environment which we want to simulate. One entry represents a calculated location (latitude and longitude) of a device (e.g. the Bluetooth beacon of a cow) at a specific time. Additionally, we keep the metadata containing details about the measurement in the table.

- **learning_measurement**

Contains preprocessed data to be used in the scenario learning component of SmartSPEC. Essentially this is the same data as in the *measurement* table, but with only the relevant fields for SmartSPEC, which are formatted accordingly.

- **location**

Contains the simulated locations, which were created by the simulator. This data is the result of a simulation.

- **smartspec_conf**

Contains the configuration of SmartSPEC. Every value is set to a reasonable default from the start. Changing the configuration affects the simulation logic.

Gateway

The gateway differs from the other logical components as it does not provide any endpoints in itself, but only forwards requests to the respective components. It serves as the central access point to the backend, preventing us from calling every service directly. In a future iteration, this would also be the point for authentication and authorization features, which are omitted in the initial prototype.

Configuration Service

The configuration service provides endpoints to handle all configurations of the simulator. In the initial prototype, there are only configurations needed regarding SmartSPEC. They are created automatically during the initialization of the simulator, so we need only endpoints to fetch them and update them. In later implementation iterations, other configuration endpoints should be added to this service.

The service uses the database for persistence and can be accessed via the gateway. It offers the following endpoints:

- **GET /configuration**

Fetches the SmartSPEC configuration from the database.

- **PUT /configuration**

Updates the SmartSPEC configuration with the provided fields. The database entries are updated.

SmartSPEC Service

The SmartSPEC service is more or less the "heart" of the backend because it is responsible for running the actual simulation. Internally it executes different components of SmartSPEC, which creates the artificial data based on the input provided. But this service is also more than just a wrapper of SmartSPEC because it also is responsible for all pre and post-processing steps of the simulation. It has endpoints to load training data which serves as input and to transform and persist simulation results as well.

The features of the SmartSPEC service can be best described by examining the endpoints it offers. We will go through them in the order users would execute them; starting with the pre-processing, then executing the simulation and in the end persisting the result:

- **GET /training-data**

Fetches the data which we want to use as input for the simulation from Track and stores it in the database. The user can define parameters like for example the start and end time

to limit the data which is fetched from Track. The request, which the user sends to the gateway, is forwarded to the SmartSPEC service. All matching measurements are then pulled from Track by the SmartSPEC service. After that, they are formatted to match the database schema and persisted in the database. The endpoint returns an HTTP (Hypertext Transfer Protocol) status code and a potential error message to the user indicating if the request was successful or not. Figure 8 describes this workflow in an activity diagram.

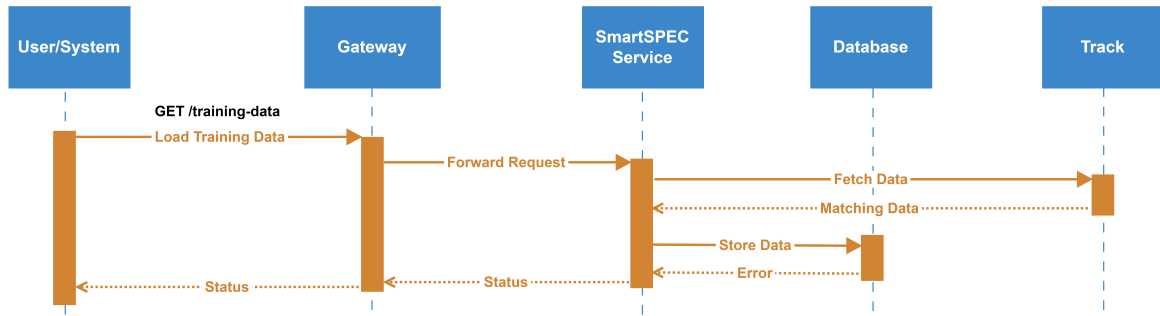


Figure 8: Workflow of the GET /training/data Endpoint.

- **POST /learning/start**

Starts the Scenario-Learning component of SmartSPEC using the configuration the user has saved in the database (cf. figure 9). After the request was forwarded by the gateway, the SmartSPEC service sends a *GET /configuration* request to the configuration service. This returns the currently stored configuration, which is fetched from the database. With the configuration in place, the SmartSPEC service starts the Scenario-Learning component of SmartSPEC in a new thread and returns the HTTP status code to the user. The learning process has to be asynchronous because it can often take multiple minutes as we evaluated in subsection 2.2.3.

- **GET /learning/status**

Returns the status of the Scenario-Learning. The possible states are: *NOT_STARTED*, *IN_PROGRESS*, *COMPLETED* and *FAILED*. Before running the Scenario-Learning it will return *NOT_STARTED*, during a run *IN_PROGRESS* and afterwards either *COMPLETED*, if it was successful, or *FAILED* on failure. Figure 9 displays the workflow of a user who first starts the Scenario-Learning and then calls the status endpoint after it is completed.

- **POST /generation/start**

Starts the Scenario-Generation component of SmartSPEC (cf. figure 10). The gateway forwards the request to the SmartSPEC service, which first checks if the Scenario-Learning was already completed and denies the request if it was not. This is necessary

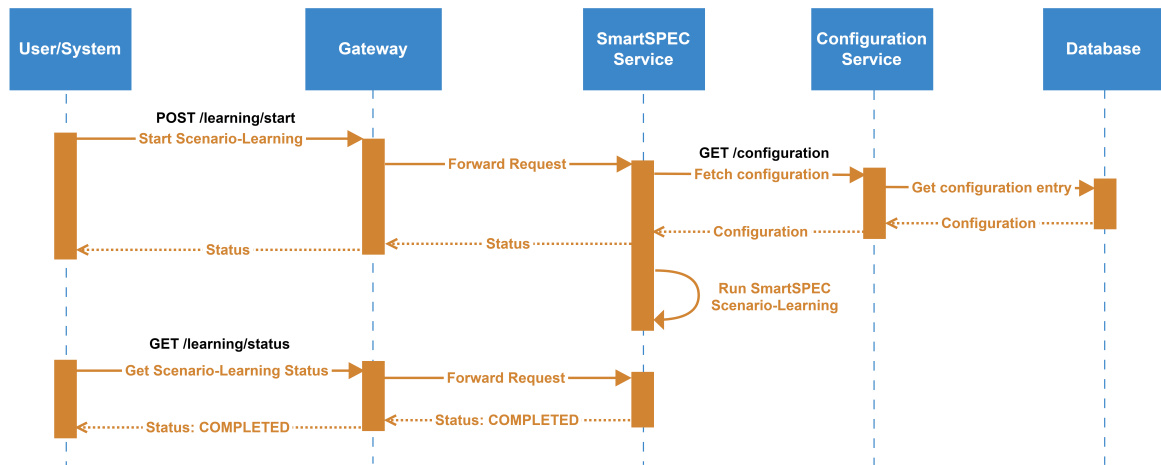


Figure 9: Scenario-Learning Workflow.

because the Scenario-Generation depends on the Scenario-Learning. After this verification, the Scenario-Generation is started. Similarly to the Scenario-Learning, this is an asynchronous process because the request is returned to the user while the Scenario-Generation is started in a separate thread. The evaluation has shown that the Scenario-Generation can also take multiple minutes according to the settings and input data.

- **POST /generation/status**

Returns the status of the Scenario-Generation. The states and logic are equivalent to the *GET /learning/status* endpoint.

- **POST /generation/persist**

Persists the simulation data, which was generated by SmartSPEC in the database. Instead of doing this in the *POST /generation/start* endpoint directly after the Scenario-Generation has finished, we decided to put this in a separate endpoint. Often we just want to try out the simulator with various settings, but not persist the result.

After the request is forwarded by the gateway, the SmartSPEC service reads in the Scenario-Generation result from its filesystem, formats it and stores it in the database. Afterwards, the HTTP status is returned to the user indicating if the request was successful. Figure 10 shows the workflow of a user who first runs the Scenario-Generation, then checks its status and in the end persists the result in the database.

Location Service

The Location Service is responsible for all requests regarding locations; in our simulator, these are the results of the simulation. In the initial prototype, we only need an endpoint to

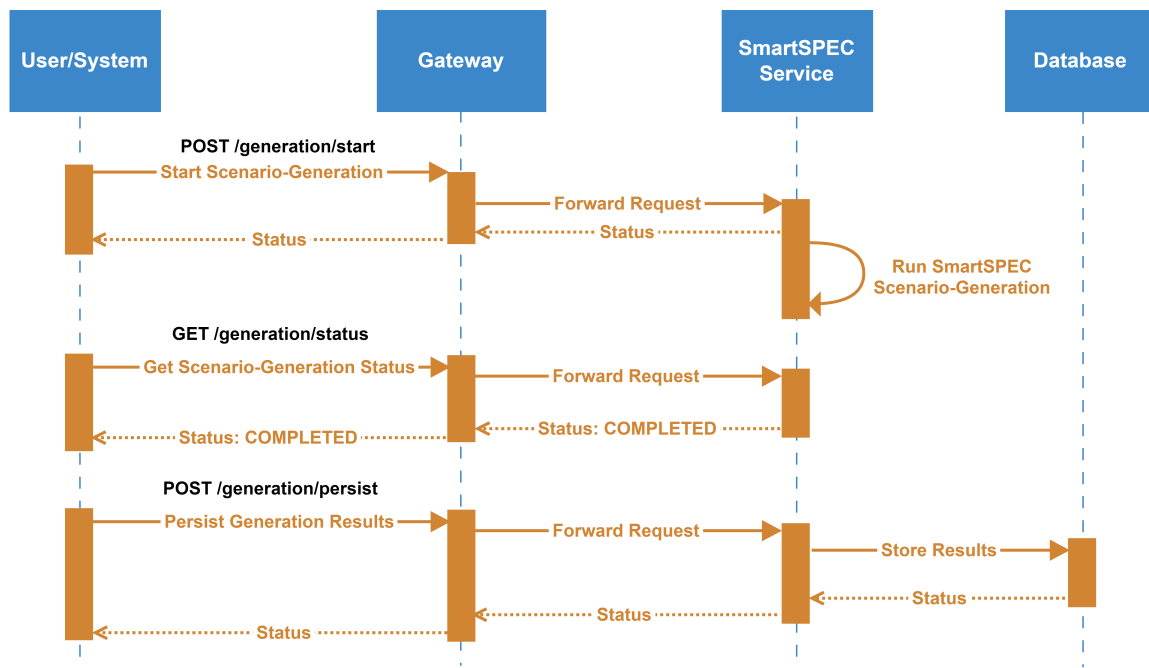


Figure 10: Scenario-Generation Workflow.

fetch the results, but in the future, this could be the service for transformation or exporting endpoints. So currently only the following endpoint is available:

- **GET /locations**

Returns the stored simulation results from the database. Based on the parameters start and end time the user can filter the results.

4.2 Frontend

The frontend is the user interface of the simulator, making it the main access point for human users. While the backend can also be used as a stand-alone system, it is designed to be connected to other systems. Requirements for human users, like straightforwardness, good readability and so on, are not fulfilled using an API directly. That is why we connect a frontend system to our backend, which fulfills these requirements to ensure the users have an easy-to-use and understandable interface to use the simulator.

Before starting the implementation a few topics have to be figured out:

- Which information has to be visible to the user?

- Which components can the frontend be divided into?
- In which environment does the frontend run?
- How should the pages be designed to fit the information?

In the upcoming contents of this chapter first the information to display is gathered and then the remaining topics regarding the environment and design are evaluated. The end product is sketches of each major element, which serve as a clear instruction plan on how to implement the frontend.

4.2.1 Information Gathering

To be able to come up with a plan for a frontend design, we first have to list down the data and information involved in the simulator, which has to be available to the user. This aligns with our approach of finding a way to visualize our known data instead of creating a good-looking visualization first and then fitting the data into it. This is the general thinking in information visualization, which is the base for any further concepts developed in this research area. One approach is to classify data into different types and define visualization rules for each type. This gives us a general framework for visualizing any data of the classified types. To achieve this, we can divide data into entities and relationships [25, pp.25-28]. This way we can pinpoint all available data and see the constraints for visualization as well. As the frontend should serve as a tool to use the entire REST API, we list down which data is emitted from it or can be provided to it. This makes up the information which the frontend has to contain:

Entities

- **Simulation Settings**

A top-level entity containing all available settings regarding the execution of the simulation, e.g. the SmartSPEC configuration.

- **User**

Information about the user of the simulator, e.g. username, last login, password hash.

- **Scenario-Learning Status**

- **Scenario-Generation Status**

- **Simulation Results**

The simulated locations.

Relationships

The Scenario-Learning Status and Scenario-Generation status are highly interrelated because Scenario-Learning needs to be always executed before the Scenario-Generation and both cannot run at the same time. Also, the Scenario-Learning status needs to be *COMPLETED* before the Scenario-Generation can be executed.

Otherwise, there are no direct relationships between the entities, only indirect ones between the endpoints. For example, the Scenario-Generation result has to be persisted in the database to be able to fetch simulation results. But these indirect relationships do not affect the design of the frontend.

4.2.2 Separation into Components

Having defined the information to visualize, including the relationships between the entities, we need to come up with a model for it. This model needs to include all the information and should separate concerns as far as possible. The relationships give an indication of which entities belong together and need to be put in one component. The result is a list of components which we later visualize separately in, for example, different pages. We explicitly do not think yet about the looks of the visualization, only about the way we structure the information. The following components could be identified:

- **Settings**

Contains the simulation settings and provides functionality to modify them using the REST API. Further settings can be added here in future iterations.

- **Execution**

Contains everything regarding the execution of the simulator. The Scenario-Learning and Scenario-Generation can be started and monitored using their statuses. The functionality to persist the results in the database is also a responsibility of this component. In the future, we could add, for example, log monitoring here.

- **Results**

Contains the results of the simulation. In the future, configurations for the visualization of the results can be added here.

- **User and Project Administration**

Contains the settings for the current user and project, including changing the email and password, resetting the project, as well as login/logout functionality.

4.2.3 Deployment Environment

After creating a model of the components which the frontend needs to contain, the next step is to think about the environment the frontend application will run in. In our case, this means deciding which type of system hosts the frontend. The options which are brought into the discussion are mobiles (smartphones or tablets), PC operating systems (OS) (e.g. Windows and MACOSX) or remote servers accessed through the web. PC OSs were excluded straight away because the use case of demonstrating and testing the real WeideInsight application is something which is mostly executed at the place where it would later be set up; a barn or pasture. This requires the frontend to be available for mobile devices and not to be fixed to a PC OS. Following on from that, the question arises if it would be the best solution to just run the frontend on the mobile devices of the users in an app. While this would also be a reasonable solution, providing the frontend remotely through the web is more flexible because it makes it device-independent. That is why we decided to design the frontend as a web page, making it accessible on any device and preventing us from deploying it to all user's devices. By making a responsive web design, the frontend is optimized for any type of device by changing the shape to fit the screen size. There we miss out on the advantages mobile apps have, like for example sending notifications to the user's phone/tablet. But for our use case, we do not see significant benefits in that.

4.2.4 Sketches

In this section, we create sketches containing each of the components defined in subsection 4.2.2. Sketching is a method of low-fidelity prototyping at the start of the development. Creating prototypes in this early stage, instead of going straight into the implementation, is crucial, as the chance of correcting errors is the highest at this point [26, p.89]. This is the case because sketches enhance mental processing capabilities for detailed problem analysis. Early specifications, as we have in our case the component model, can be tested by trialing a design without any functionality. In historic times, sketches were drawn by hand on paper; these days often software is used which eases this process and provides reusable components. We will use Excalidraw to sketch the frontend, which is an easy-to-use drawing tool to create "hand-drawn" sketches [27].

Navigation Bar

This sketch, visible in figure 11, defines the navigation bar which is visible on every page of the frontend. Being placed as a fixed element on the top of the page, it contains buttons to allow the user to navigate between the pages of the frontend. From left to right, it starts with the name/logo of the simulator (we called it *SimCattle*), which takes the user on click to the execution page, which is so to say the "main" page. Next up follow links to all the pages

in the frontend, *Execution*, *Map View* and *Settings*. On the right-hand side, we placed the user icon, which opens up a dropdown on click containing the tasks of the *User and Project Administration* component.

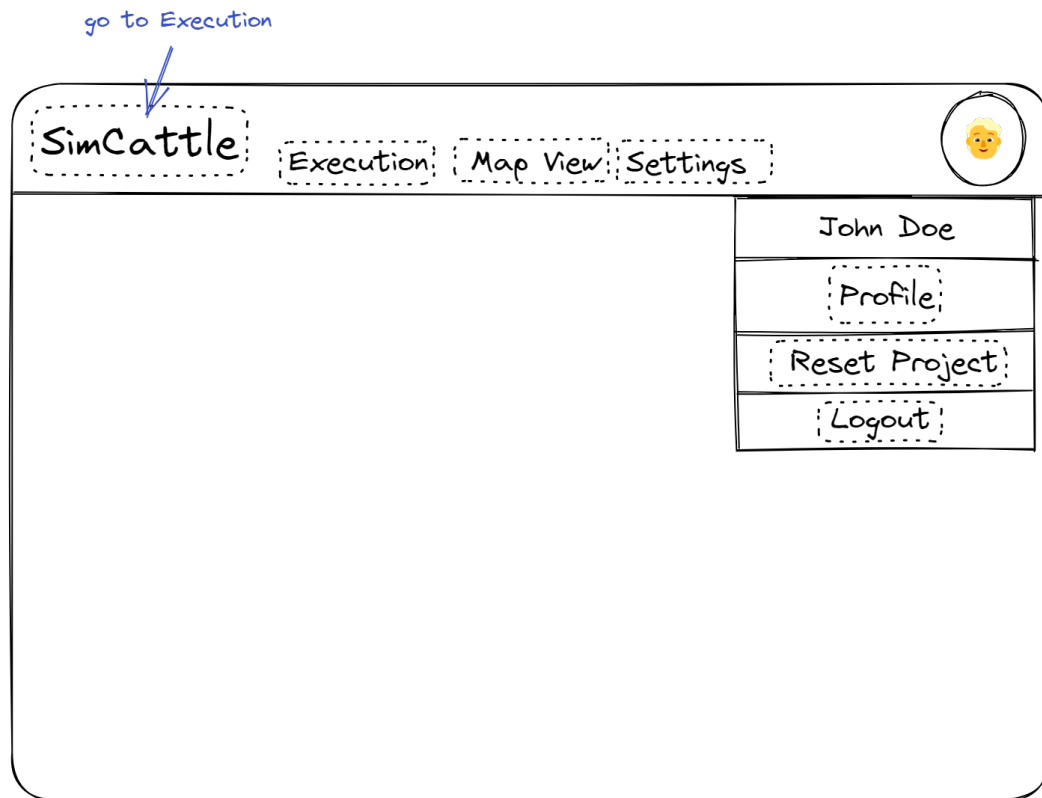


Figure 11: Design-Sketch of Navigation Bar.

Execution Page

The execution page, which serves as the main and initial page upon opening the frontend is sketched in figure 12. It contains all information of the *Execution* component. The challenge here is to display the pipeline-like flow of creating a simulation. A simulation starts with the Scenario-Learning, after which the Scenario-Generation can be executed. The result of this then needs to be persisted in the database if so desired. In the sketch, each step is displayed on a card, which contains all information related to the specific step. This information includes the title, the description, a button to execute it and optionally the status. Seeing each step as a separate component adjusted with parameters will help us later in the implementation by creating a reusable component *Step*.

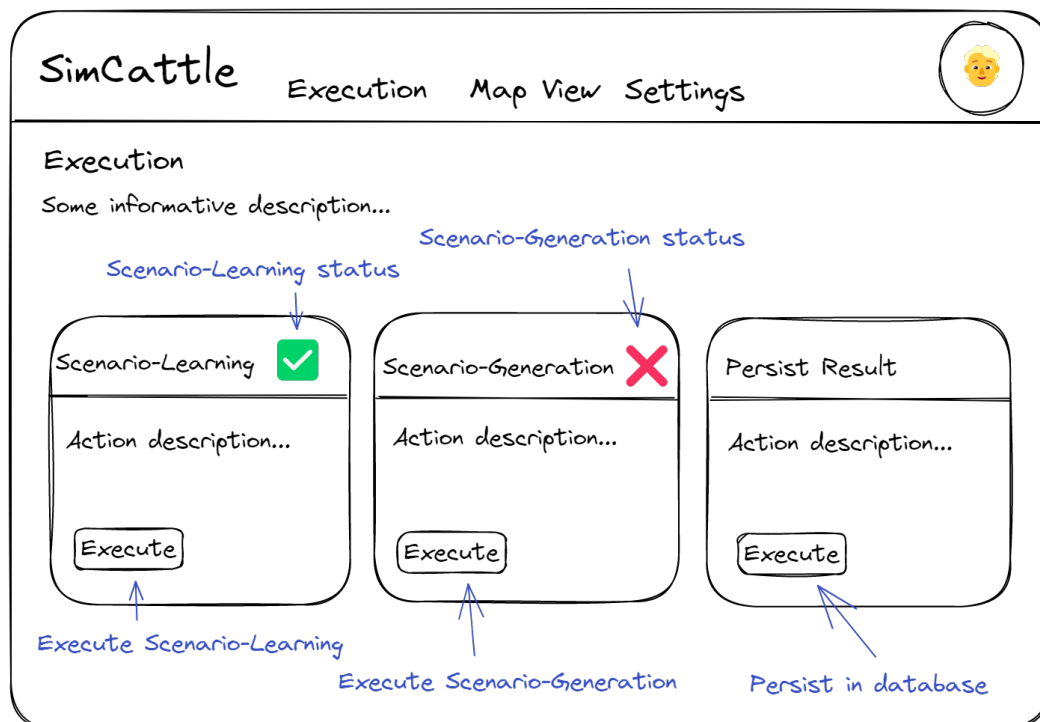


Figure 12: Design-Sketch of Execution Page.

Map View

Map View is the page containing the *Results* component. Here the results of the simulation are shown on a map. This is challenging because of the spatio-temporal characteristics of the results: Each result is a location of an entity at certain coordinates in the period from its start time to its end time. As this data has three dimensions, latitude, longitude and time, it is non-trivial to display it in a two-dimensional environment like our web page. There is a way to display the data all at once in three dimensions using a Space Time Cube (STC), but that seems too unconventional for our use case [28]. Rather we solve this with the use of an animation which increases the time periodically to show the data in a timely order. This also resembles the actual WeideInsight system, which we want to simulate. The user gets the impression of a live view of the data, just like he would see in reality. The animation can be paused/resumed using a button on the top left of the map. On the right, we see the current simulation time. The bottom right is reserved for action buttons related to the map, like editing the layout or setting markers.

Settings

The *Settings* page is used to display and edit the information of the *Settings* component. As shown in the figure 14, the design is simplistic, having a title and a form field for each

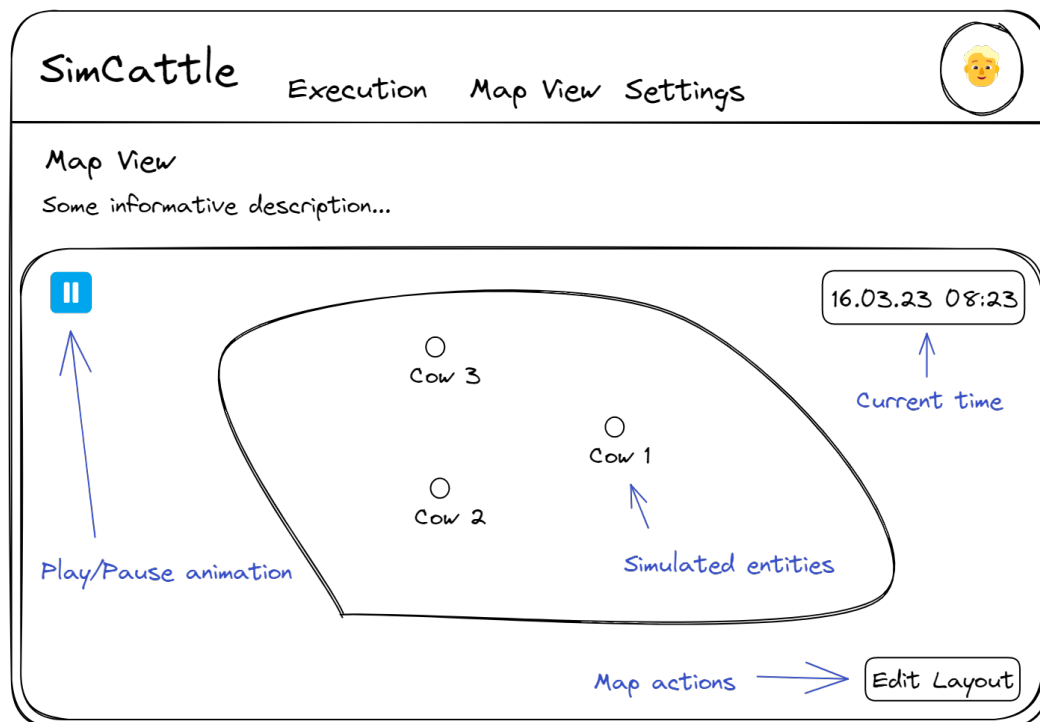


Figure 13: Design-Sketch of Map View Page.

setting. As the settings are highly technical in our case, this page needs a bigger amount of documentation. That is why each setting receives a "?" button, which opens up a modal containing more documentation about the setting. It is also important to link to the official SmartSPEC documentation here, wherever it is useful.

To conclude: In this section, we sketched the main functionality of the frontend with little detail to judge if our defined components are sensible. This is helpful, as flaws in the model were easily identified and fixed during sketching. This concludes the architectural setup of the frontend, which guides the implementation afterwards.

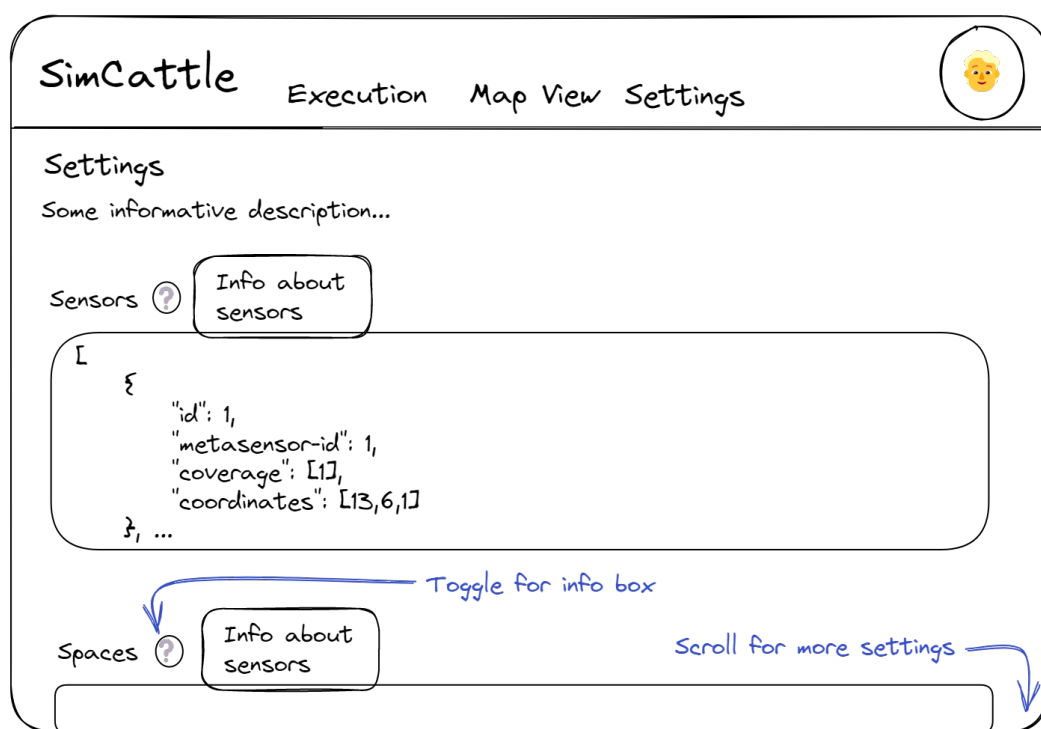


Figure 14: Design-Sketch of Settings Page.

5 Implementation

This chapter contains all matters regarding the implementation of the simulator. The implementation brings together the previous work of the chapters 2, 3 and 4 to create a functioning program out of the ideas, requirements and architectural models we defined.

5.1 Backend

After we previously defined the functionality of each service in chapter 4, we are ready to implement the services. Before that, we will give justifications for all decisions relevant to the implementation. Then we will examine the code structure for a better understanding of what files a service internally consists of. For the logical part of the service, we will present the pattern which we use to separate concerns as far as possible. A special look is taken into the SmartSPEC service, as it is the main service of the backend. We conclude with a usage guide for the implemented REST API.

5.1.1 Design Decisions

The chapter 4 already hinted towards a few design decisions done in the implementation, e.g. the usage of Docker and Go. We want to give an explanation of why the chosen technologies are ideal for our use case and compare them with alternatives.

Docker and Docker Compose

As we want to build a microservice architecture with various services, a gateway and a database, we need a smart way to run them. What also has to be considered is, that the services communicate with each other. The environment where we run the services has to support this preferably in an easy way. We decided to use Docker because it helps us in multiple ways during implementation and for the deployment of the simulator [29]. Running each service in a separate Docker container encapsulates the services to different processes which are independent of each other. It also makes it easy to distribute the simulator to customers and stakeholders, as Docker images are platform-independent and require only a Docker instance to run on any machine.

To orchestrate a flexible amount of Docker containers we use Docker Compose [30]. This tool allows us to build all images, as well as start and stop all containers in one command. Furthermore, it adds all containers to a separate network, which is the communication channel between the containers. Simply defining a *docker-compose.yaml* file listing all the containers and parameters is enough to orchestrate the containers in a simplified way.

There is no equally convenient solution currently to run microservices using Docker available on the market, so the decision here was easy. Although many other solutions exist for the orchestration of containers, for example, Docker Swarm or Kubernetes, these technologies have much higher capabilities than Docker Compose, like for example replicas, auto-scaling and load balancing. But for our use case, they would only over-complicate things, as Docker Compose offers everything we need by only defining one configuration file.

Go

To program the logical microservices we chose the language Go, also known as Golang [24]. This is not an as straightforward decision as it is to use Docker, because there exist many suitable languages for this architecture.

The reasons for this can be found in the history of why Go was created: Three Google developers, Rob Pike, Robert Griesemer and Ken Thompson, concluded that the conventional languages the Google platform was based on at the time, i.a. Java, could not handle the increasing demand for concurrent multi-core applications anymore. That is why they decided to implement the new language Go in 2008, which has a much faster compile time and higher efficiency than other statically typed languages. Even though Go is statically typed, this is not as obvious when writing the code, as many redundant statements, as Java has, are omitted. As an example, we can compare how to instantiate an object in both languages (cf. listing 1). The redundant double specification of the class and the keyword *new* are left out in Go.

```
// Java
MyClass someObject = new MyClass()

// Go
someObject := MyClass{}
```

Listing 1: Java vs. Go: Class Instantiation

Go is also optimized for concurrent programming [23, p.26-27]. The language was developed to help Google create efficient microservice architectures, exactly like we need it in our use case [23, p.34-35]. That is why the decision was made to use Go instead of a conventional language like Java.

MySQL

In the persistence layer of the simulator, we have the option to choose between a relational (SQL) or non-relational (NoSQL) database. Non-relational databases, especially time-series databases like InfluxDB have their strengths in storing unstructured and high-quantity data.

This would fit our requirements because the training and simulation data used in the simulator fulfill these criteria. But it has to be considered, that also much different data like configurations should be stored in the same database. It is also not required that the data is stored quickly because this is anyway done by asynchronous processes. The query speed is a factor to consider, but it is only marginally different for SQL and NoSQL databases with the amount of data we have.

Because much of the data in the simulator is relational and there are no significant advantages in using a NoSQL database, it was decided to use a SQL database. We chose MySQL out of the most popular options because it is easy to integrate into a Docker environment.

5.1.2 Code Structure

In every service of the simulator backend, Go is used as the programming language. Additionally, every service builds up from a simplistic boilerplate service, which has the general functionality every service uses already implemented. This includes a main file to start the service, the database connection handler, the routing setup and a general error handler. The boilerplate also defines the file structure inside the service making every service organized the same way. This approach might seem unconventional for people insisting on fundamental programming concepts like the "Don't Repeat Yourself" (DRY) principle. These concepts are still applicable for microservice architectures but are limited to each microservice, not across the whole application. Using our boilerplate service as a base for all services most code, excluding the actual logic of the endpoints, is copied in each service. Instead of insisting on the DRY principle, we prefer to keep the services completely autonomous. This aligns with the principles of microservice architectures and brings in the advantages of distributed development and fault isolation [31, chap.1]. Another approach would be to develop a common package and import it into every service, instead of repeating the code. Then the package would be a central component and faults inside it would affect the whole system. Distributing development to the respective services isolates the faults. This advantage is at cost of the DRY principle but was proven to be ideal for big-scale applications.

Having defined the boilerplate and concept of how to structure a service, we can now have a look at how a completed service is structured. Inside a service, our goal is to separate concerns as much as possible, in a similar way as we do across the services. As an example, we will use the location service, because it is in the initial implementation iteration very minimal, offering only one endpoint. Figure 15 shows all folders and files inside the service with short summaries of the contents. In principle, the service is divided into different packages, which are represented as folders in the file structure. Each package has different responsibilities:

- **Main package**

Includes all files in the root directory. The *Dockerfile* specifies the commands for building a docker image of the service. The *go.mod* and *go.sum* files contain the packages which are used in the service. *main.go* is the file which is executed to start the service.

- **.vscode**

Folder for configuration of the VS Code development environment. Contains the configuration to run and debug the service.

- **Api package**

Contains all necessities to make the service offer a REST API. This is mainly a router, which accepts requests and forwards them to the correct function according to the route provided by the caller. This router listens at a defined port for requests.

- **Db package**

Contains the database handler, which creates a connection to the database on startup and initializes a client to access it.

- **Errors package**

As the default *error* class in Go only contains an error message and no status code, we use a custom wrapper class to also pass around a specified status code.

- **Http package**

Contains *.http* files to make requests to the API for testing purposes. The IDE VS Code supports these files and is comparable in functionality to Postman, for example.

- **Location package**

The package contains the actual logic of the endpoints regarding *locations*. Confer subsection 5.1.3 for details.

5.1.3 Pattern for the Service Logic

While everything which can be generalised is part of the boilerplate service and copied in each service, the logic for the endpoints differs according to their purpose. Nevertheless, we can also standardise the logic by splitting it into layers, which serve a certain role. Such a layer structure can then be applied in every service. There exist many popular patterns to organise code into layers, for example, the Model-View-Controller (MVC) pattern [32]. In our case, an MVC pattern does not make sense, as the view part is split into a separate frontend application. Instead of the backend returning HTML, which defines how the frontend looks like, our backend returns JSON which only contains data and no information about how it is visualized. We identified the Controller-Service-Repository (CSR) pattern to be suitable

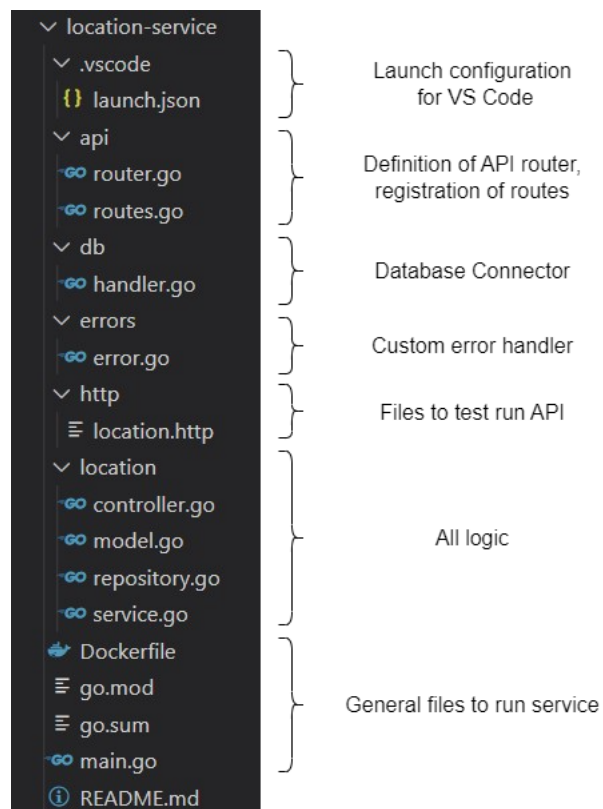


Figure 15: Code-Structure of Location Service.

for our needs [32]. This pattern splits the logic into the three layers, controller, service and repository (cf. figure 16). Each of them serves a different role to separate concerns.

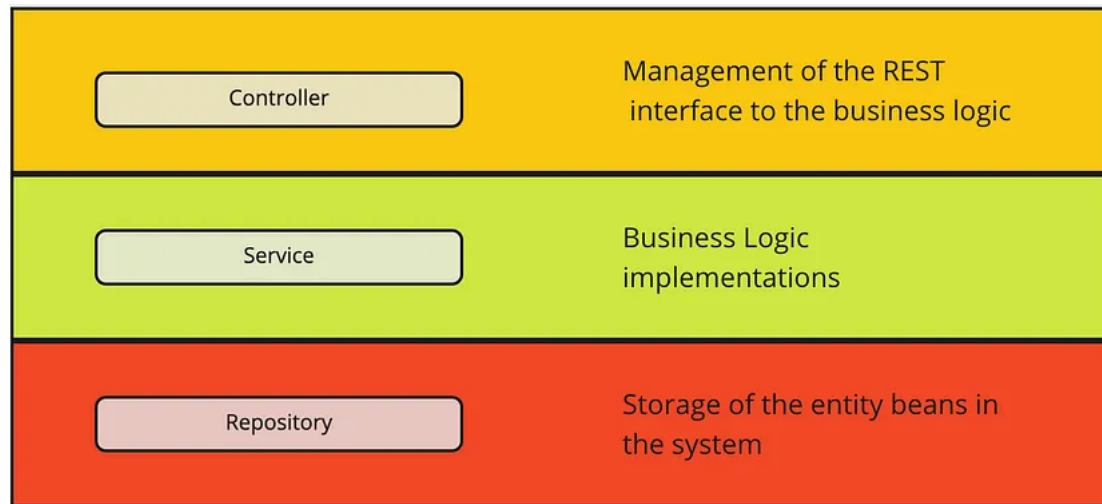


Figure 16: CSR Pattern [32].

Applying this pattern to our backend, results in the following responsibilities of each layer:

- **Controller**

Contains the functions which are directly called by the router. They are so to say our REST endpoints, as they are the functions executed when calling an endpoint. Its main responsibilities are to bind and verify the parameters sent along with the request, to call the service layer with the parameters and to respond with a status code and message. The controller functions explicitly do not implement any logic and serve only as the access point for the REST API.

- **Service**

The service layer contains all the logic of the endpoints. This can be any processing tasks, accessing other systems or logging. The database is never accessed directly, only through the repository layer.

- **Repository**

The repository layer is responsible for accessing the database. This can be simple read/write operations, but also complex transactions with multiple requests to the database. The repository is called only by the service.

The beauty of the CSR pattern is, that each layer only knows about the things it needs to know. The repository layer works independently and holds no information about the existence of a service and a controller above. Similarly, the service only knows the repository

and not the controller or, in the other direction, the database. Every layer only knows the layer below. This makes each layer easily reusable and replaceable.

5.1.4 Logic of the SmartSPEC service

As an example and because it is the main service of the backend, we will have look at the internal logic of the SmartSPEC service. From a top-level view, this service wraps the SmartSPEC software in a service and enhances it with additional functionality. The two components of SmartSPEC, Scenario-Learning and Scenario-Generation, are being made available as a service. This requires us to run them both from inside the service, in particular from the *service.go* file. As the Scenario-Learning is written in Python and the Scenario-Generation in C++, we decided to call both components via bash commands, rather than installing connectors for these languages in Go. The *service.go* file runs bash commands to access both of the components which are stored in the file system of the service. SmartSPEC stores the results on the file system as well during execution. *service.go* reads in these files for persisting the result in the end. In simple words: The SmartSPEC service does all the tasks which normally would be manual effort when using SmartSPEC.

5.1.5 Usage Guide

This section presents the functionality of the backend with a step-by-step guide on how to create and use a simulation using the REST API.

Prerequisites

- Docker
- make
- Visual Studio Code (optionally for making requests to the REST API)

Start the Backend

You first have to clone the thesis repository, if not done already (cf. appendix A). Then navigate to the *code* folder inside the thesis repository and run *make up* to start all services:

```
cd 2022-ma-paul-pongartz/code
make up
```

After this, all services should be running, which you can confirm with the command *docker ps*.

Prepare the Training Data

By default, there is ground truth data from a previous experiment in the WeideInsight project stored in the database. If you want to use a different data set, you need to insert it manually in the *measurement* table in the database. Then use the script *code/transform_measurements.sql* to prepare the data for the use in SmartSPEC.

Adjust the Settings

There are default settings provided, which can be fetched from the API using the following endpoint:

GET <http://localhost:8080/api/configuration>

There are helper files in the *http* folder of the source code of the respective service to execute the requests easily inside Visual Studio Code. Alternatively, you can use any other tool, e.g. cURL or Postman.

You can adjust the settings by calling the endpoint:

POST <http://localhost:8080/api/configuration>

Specify the settings you want to change in the body of the request. Confer the *http* file for examples.

Execute Scenario-Learning

To start Scenario-Learning, which builds a model based on the training data, call this endpoint:

POST <http://localhost:8080/api/learning/start>

Depending on the settings this can take a few seconds or several minutes. You can check the status by calling:

GET <http://localhost:8080/api/learning/status>

Execute Scenario-Generation

After Scenario-Learning is successfully completed, you can start Scenario-Generation by calling:

POST <http://localhost:8080/api/generation/start>

This will create simulated locations using the model created in the Scenario-Learning step. There is also a status endpoint for Scenario-Learning:

GET `http://localhost:8080/api/generation/status`

Persist the Result

After successful completion of Scenario-Generation, you can persist the simulation result, which are locations, from the internal file system of the service to the database by calling:

POST `http://localhost:8080/api/generation/persist`

Fetch the Results

After the results are persisted you can query them by calling:

GET `http://localhost:8080/api/locations?start=<timestamp>&end=<timestamp>`

The two query parameters *start* and *end* can be used to limit the locations to a certain time frame. This refers to the simulated time and returns only locations with a timestamp in this range.

5.2 Frontend

After we already got an idea about the components and the looks our frontend should have in chapter 4, we now have to find a way to implement them. In comparison to the backend, we were able to define even more details in the architecture, for example using sketches. This reduces the thinking process during the implementation and allows us to focus on a good code design.

We will operate similarly to the backend implementation and first give justifications for the decisions made regarding the programming language, framework and other technicalities. Then the code structure is examined and verified using the component model from chapter 4. In the end follows a usage guide for the Web UI we designed, which includes screenshots of the implemented pages.

5.2.1 Design Decisions

In chapter 4 we decided to implement a Web UI, instead of a mobile app or a PC software. There are various technologies, languages and frameworks available to accomplish that. This

section will give reasoning why we chose to implement the Web UI using the JavaScript framework React. This includes tools which help us write React code (so-called "frameworks for a framework").

JavaScript

While the web is based on HTML and CSS, there exist many possibilities to generate, modify and interact with these definition languages. While it is possible to do that using PHP or Python, most people choose different languages, because they offer more capabilities. According to a recent study, JavaScript ranks top in the list, with Python and Typescript following up [33]. This is not a new trend, as JavaScript is by far the most used web development language for centuries. Newer ones like Typescript have the potential to take the top spot, but there is a factor which currently makes JavaScript stand out: There exist countless frameworks built on top of JavaScript, more than any other language. The top six web frameworks in 2022 are using JavaScript internally [34]. This makes JavaScript suitable for any type of web application, also our simulator frontend.

React

React is a web framework for JavaScript designed to create user interfaces [35]. It allows us to split a web page into reusable components, which compile into a piece of HTML code each. To define a component we can use default JavaScript syntax in combination with a definition language called JavaScript XML (JSX). JSX is based on HTML and extends it with capabilities to insert anything JavaScript related dynamically. The name suggests that the framework is "reactive", which means that the created program is expressed as a reaction to its inputs, through which data flows [36, Chap.2.1]. In easy words, this means React produces a static web page, which can be modified using events. There is much equal competition to React, for example, Angular or VueJS, which have similar scopes of application. React is not particularly suited better for our use case than the alternatives and was chosen out of a range of equal options.

React Bootstrap

React makes it easier to write web applications, but it defines no design components itself. This means that without additional tools every visible element in the browser has to be defined using HTML and CSS. Gladly there exist frameworks like Bootstrap, which are collections of standardised HTML elements like for example buttons or cards. The regular bootstrap framework is not compatible with React and its JSX definitions, so there exists a React Bootstrap framework which accomplishes this connection [37]. We will use this framework to use regular bootstrap components in our web pages.

Axios

Apart from the visual side, our React application also needs to use the REST API to interact with the backend. React provides an internal REST API client using the function *fetch*, which is quite simplistic and does not do much automation. The Axios framework is an enhanced version of *fetch* and eases the development process of REST API connections [38]. In contrast to *fetch* it is promise-based, which means that any API call is by default executed in the background in an asynchronous process. It also automatically converts JSON strings to objects and in reverse, which prevents us from writing our own marshalling/unmarshalling functionality. That is why we chose to use Axios for the connection to the backend.

5.2.2 Code Structure

In comparison to the backend, the internal complexity of the frontend is much easier to grasp, as there is only a single application and not multiple independent microservices. Nevertheless, we also try to separate concerns in the frontend by using React's component capabilities. The initial idea is to separate view logic from the data processing, in our case the visual pages from the REST API connection. Continuing from that, we split the view into different components. Every page we defined in the sketches should be its own component and referenced in the global navigation bar. In the next step, we can look into each page and see where it makes sense to create further sub-components and embed them in the page. Approaching the implementation like this ensures that components do not get too big, which leads to bad readability, unwanted dependencies and further problems. If possible, the components can then be reused at multiple places which unifies the pages to a common shape.

We implemented the frontend applying this approach. The result, from a technical side, can be seen in the created code structure shown in figure 17. The code is split into the following parts:

- **Components**

The folder contains all components from all pages. Every sub-folder is dedicated to one bigger component. These are first of all the separate pages *Execution*, *Map* and *Settings* and second of all the *Navbar* and *Toasts* components which are used on every page. Each component is defined by a JavaScript file and a CSS file. The page components have further sub-components for specific elements in the page. For example, the *Execution* component has a sub-component *Step*. This is a generic component used for every pipeline step.

- **Services**

Contains the connectors to each endpoint in the REST API. They are split into three files representing the three services to improve readability. Each endpoint call is wrapped in a

function which extracts the data out of the result and returns it. These functions are used in the respective places in the components.

- **Utils**

Contains code which is needed in multiple places, but does not belong in the services or components category. Currently, this folder only contains an initialisation of the Axios client, which is used in the services to connect to the REST API.

- **Global elements**

The root folder of the application source code contains all global elements. This includes the root of the React application and the router settings. The overall layout of the Web UI, having a navigation bar at the top and below the respective page, is defined in *App.js*.

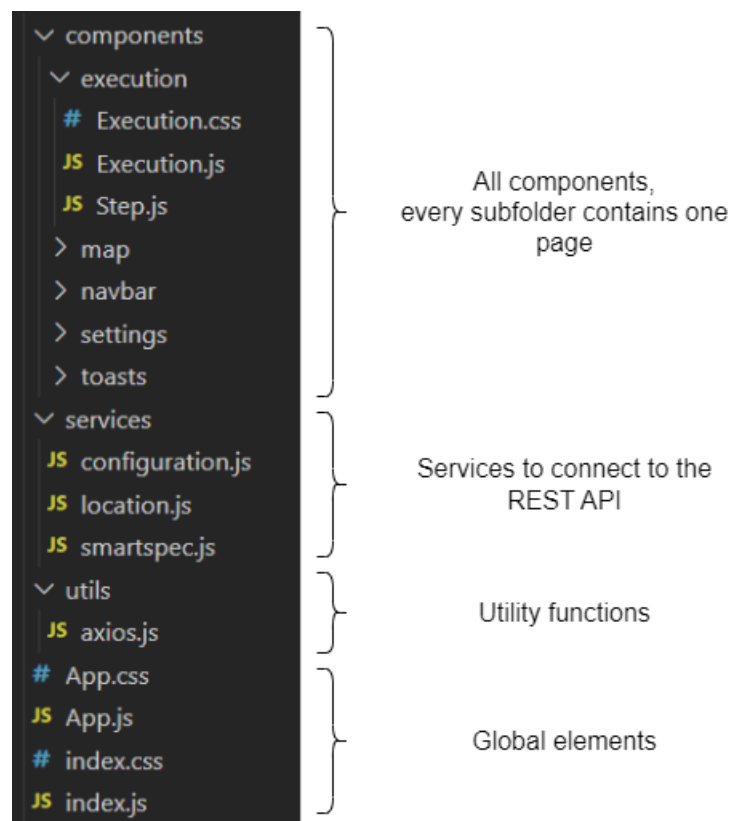


Figure 17: Code-Structure of Frontend React Application.

5.2.3 Usage Guide

In this section, we want to present the functionality of the Web UI by providing a usage guide which takes us through a complete workflow to create a new simulation.

Installation

Refer to the usage guide of the backend in section 5.1.5 to start the simulator and prepare the training data. After running *make up* the Web UI is available at **http://localhost:8080/**.

Adjust the Settings

After accessing the Web UI at **http://localhost:8080/** you should see the *Execution* page by default. Before executing the simulator you might edit the default settings by navigating to the *Settings* page in the top navigation bar. The settings page should appear as shown in figure 18. To get more information about a specific setting press the ? next to it. At the bottom of the page, you find a *Save* button, which persists the changes you did in the form. A popup message should appear on the top right indicating a successful save.

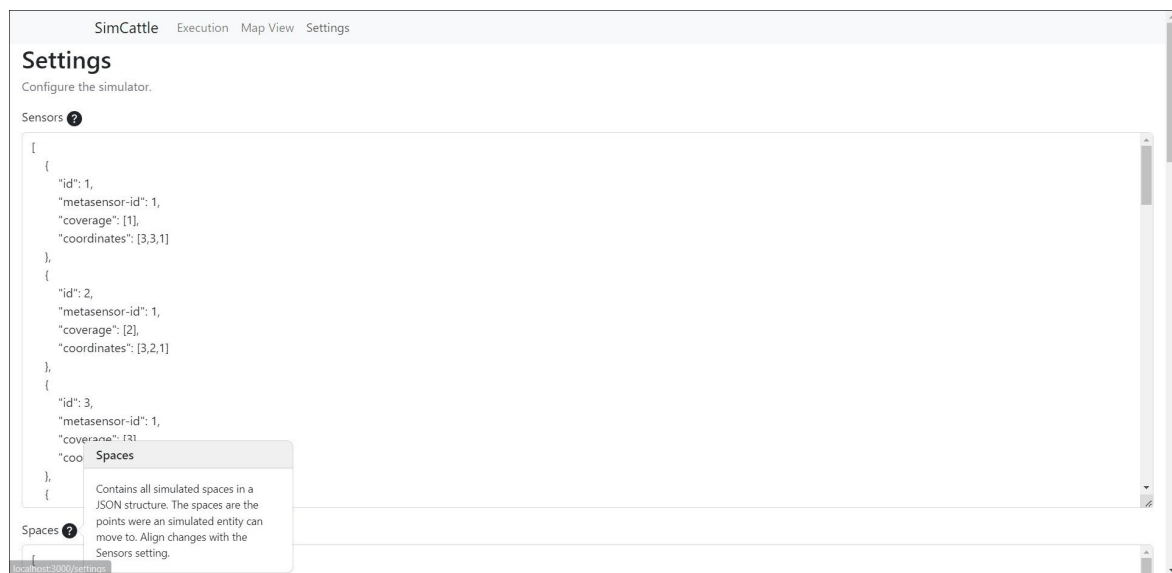


Figure 18: Web UI Page: Settings.

Execute Scenario-Learning

Go to the *Execution* page using the navigation bar. The page should appear as shown in figure 19. Run Scenario-Learning by pressing the *Start Scenario-Learning* button on the left card. The status should switch to *In Progress*. Wait till the status switches to *Completed* before progressing. This could take several minutes depending on the settings.

Execute Scenario-Generation

If not already there, navigate to the *Execution* page. If Scenario-Learning has the status *Completed* you should be able to execute Scenario-Generation in the middle card. Wait

till it has the state *Completed* as well before progressing. This could take several minutes depending on the settings.

Persist the Result

After Scenario-Generation received the *Completed* status you can persist the result to the database in the right card. A popup message indicates if this was successful.

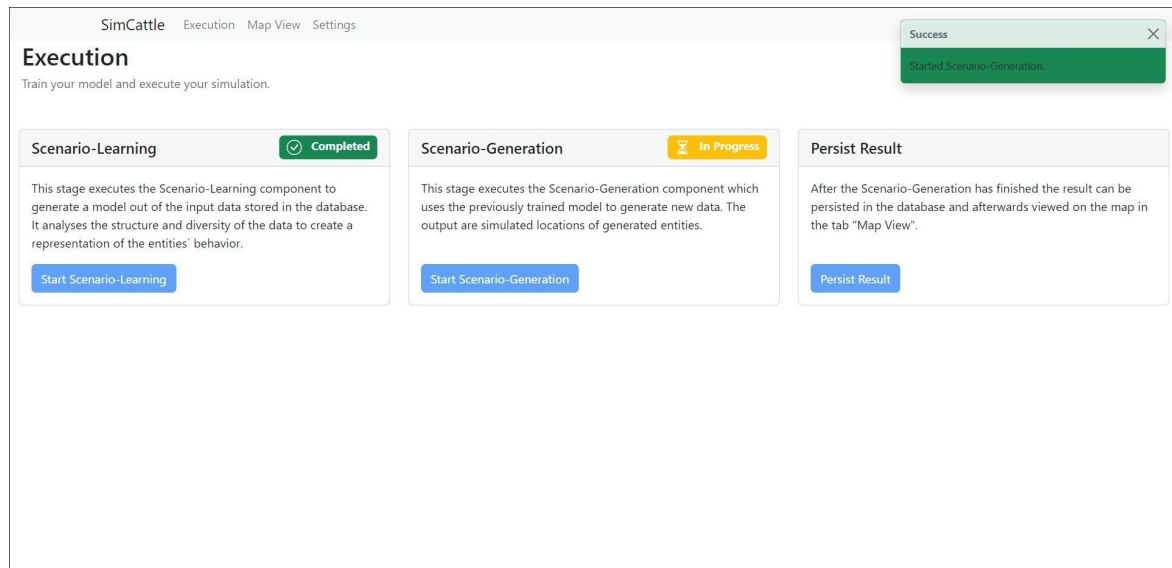


Figure 19: Web UI Page: Execution.

View the Results

After persisting the result you can navigate to the *Map View* page and view the results on the map. The page should look like shown in figure 20 and have entities displayed as blue dots and the sensors as bluetooth icons. By default, the animation is enabled and you can see the time progress in the top right of the map. To pause it press the button in the top left. You can specify a layout, e.g. the borders of the barn, by pressing *Edit Layout* in the bottom right of the map. This activates the drawing mode on the map and you can click together a layout and save it.

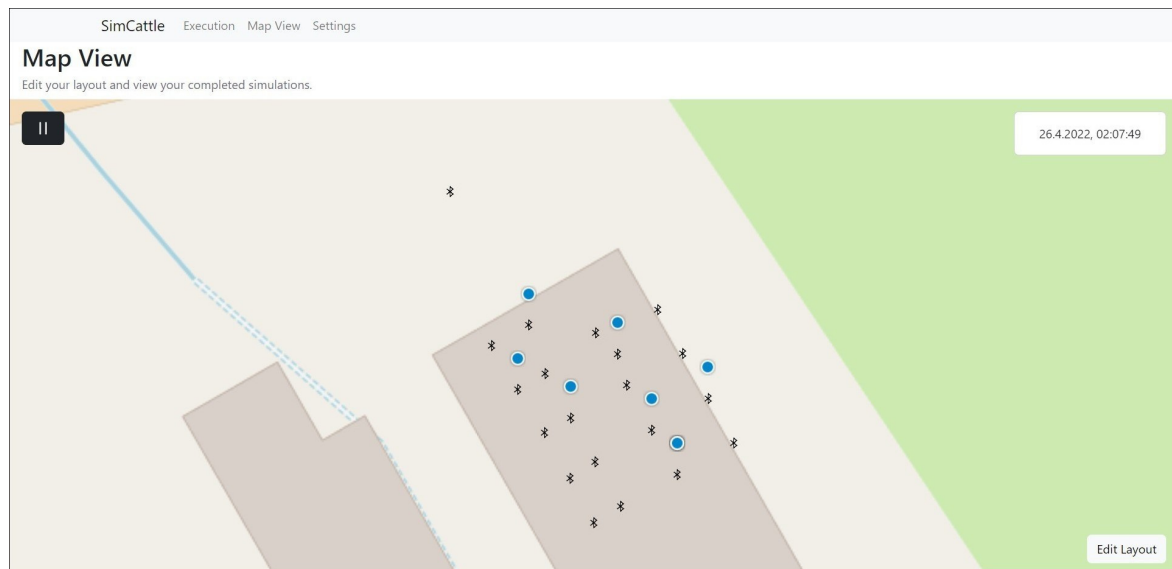


Figure 20: Web UI Page: Map View.

6 Evaluation

In this chapter, we will evaluate the functionality of the simulator using the front- and back-end according to realisticness and usability. This judicial process verifies if our implementation has value and can fulfill the requirements or if more implementation iterations are needed. Sometimes this can also lead to a radical shift in the approach to the problem if the results are not sufficient to solve it.

Our approach to evaluate the simulator is split into two parts: First, we run an experiment trying out different settings to find out the ones, which produce the most realistic results. This is judged by analysing each test case's result statistically to bring meaning to the location data. Secondly, we try to create an accurate simulation with the same characteristics as the ground truth data to prove that the simulator is capable of accomplishing this. In the end, we check if the requirements were fulfilled based on the results of the two performed experiments.

6.1 Experiment 1: Testing Different Settings

The goal of this experiment is to find out how the simulator can produce results using different settings. We use a fixed set of ground truth data as input for Scenario-Learning and also fixed spaces, sensors and metasensors settings. In contrast, we diversify the SmartSPEC configuration files for Scenario-Learning and Scenario-Generation. After collecting different results we assess the difference between them by statistical means. To conclude, we evaluate if there are "good" and "bad" settings, meaning how well the settings reflect the actual behaviour of the simulator.

6.1.1 Setup

With the idea in mind to test a vast amount of settings to assess which ones produce good results, we can define a list of test cases to run. Because of the big amount of variables to adjust in the simulator, it is not possible to test every possibility. That is why we define some variables as constant and diversify only the ones which affect the result the most. Let us go through the constants and variables for the experiment:

Constants

- **Training data**

We use measurements from a previous field experiment conducted as part of the WeideIn-

sight project. In a time frame of three days ten cattle were surveilled by twenty Bluetooth sensors in a barn.

- **Sensors**

The sensors setting resembles the actual sensors of the ground truth data. We adopted the real coordinates and neighbour sensors.

- **Spaces**

We define a space for every sensor in the barn. Each space has the same coordinates as the respective sensor, meaning we copy the coordinates from the sensors setting. This was done for simplicity, future work could explore how adjustments here would affect the simulation.

- **Metasensors**

In the field experiment, only Bluetooth beacons were used, so we defined this as the only sensor type.

- **Start and end time**

The earliest data is measured on 26 April 2022 and the latest on 28 April 2022. For Scenario-Learning we kept these dates the same to learn from all available data. For Scenario-Generation we reduced it to 26 April only to limit the simulation to a period of 24 hours. As SmartSPEC allows only dates for the start and end times, this is the minimum period we can simulate.

Variables

- **Generation and number of entities**

In Scenario-Generation we can adjust how entities are generated. As SmartSPEC was initially designed for the simulation of humans, the entities are referred to as *people* in the settings. First of all, you can select using the field *generation* where the entities come from. If it is *none*, it just takes the learned entities from Scenario-Learning, if it is *all* then all entities are generated based on meta information from Scenario-Learning, if it is *diff* some entities are generated and some taken directly from Scenario-Learning. Second of all the number of entities to create is set in the field *number*. When *generation* is set to *none* this field is skipped.

- **Generation and number of events**

The same *generation* setting as there is for the entities is also available for the *events*. This works similarly and you can specify which information to pass from Scenario-Learning to Scenario-Generation.

- **Smoothing function**

The smoothing function to apply on the occupancy graphs during Scenario-Learning.

This, for example, prevents all entities from being in the same spot. The smoothing makes sure all spaces are covered equally on average. The options are Simple Moving Average (*SMA*) and Exponential Moving Average (*EMA*), which we will diversify. *SMA* averages the values in a specified *window* to smoothen potential outliers. *EMA* enhances *SMA* by applying more weight to recent data in the *window* than older one. Both algorithms are resulting in a "trend" of the data, while *EMA* reacts faster to changes by weighting the newer data more [39]. Using the setting *window*, we can specify the window size for the smoothing function. We will leave at a fixed default of 10 minutes.

Having the variables and constants set up, we can define the test cases. The idea is to diversify the settings to cover at least two different values for each variable. For the fields *generation* and *smooth* we can try out every setting, as they can only have two and three different possibilities respectively. The *number* fields for both *people* and *events* can be any integer, so we have to decide on two different values each. For the *people*, which are in our case the entities, the values 10 and 50 are chosen to represent a little and bigger amount of entities. For the events, we have to remember, that the *number* indicates all occurring events, which are basically movements of multiple entities, in a day. That is why we decided on the values 1000, which equals an event every 86 seconds, and 3000, which equals an event every 29 seconds.

Table 3 contains the test cases, which we will execute, along with the variable settings. A complete set of settings including all definitions, which might not be mentioned here, is listed in appendix B. They are the settings for the test case 1.

Test Case	smooth	people: generation	people: number	events: generation	events: number
1	EMA	all	10	all	3000
2	EMA	all	10	all	1000
3	EMA	all	50	all	3000
4	EMA	all	50	all	1000
5	EMA	diff	10	diff	3000
6	EMA	diff	10	diff	1000
7	EMA	diff	50	diff	3000
8	EMA	diff	50	diff	1000
9	EMA	none	-	none	-
10	SMA	all	10	all	3000
11	SMA	all	10	all	1000
12	SMA	all	50	all	3000
13	SMA	all	50	all	1000
14	SMA	diff	10	diff	3000
15	SMA	diff	10	diff	1000
16	SMA	diff	50	diff	3000
17	SMA	diff	50	diff	1000
18	SMA	none	-	none	-

Table 3: Test Cases for Experiment 1.

6.1.2 Execution & Results

The execution of each test case results in a set of simulated locations, each having coordinates, a start and end time, a *space_id* and an *entity_id*. We can view the locations on the map in the "Map View" tab in the frontend which is an initial indicator of the data quality. But as this is a subjective first picture, it is necessary to evaluate statistics proving the quality mathematically. We decided to compare the test case results by using the following statistics. They are ordered by complexity:

- **Total locations**

The total amount of records in the simulation result.

- **Number of distinct entities**

The number of distinct entities present in the simulation.

- **Number of covered spaces**

The number of spaces which were visited at least once. We defined 19 spaces, so this is the maximum.

- **Most visited space**

The space which was visited the most by any entity.

- **Total space movements**

The number of times any entity changed its location, which means moving to another space.

- **Average spaces moved per entity**

Average times an entity moved to another space.

- **Average distance moved per entity**

The average distance an entity moved in kilometres (KM).

Table 4 contains the statistical results of each test case defined in table 3. For the most part, the results resemble the expectations, proving that the settings work as they are supposed to. Nevertheless, we can extract some interesting facts.

- Every test case covered only 16 spaces, which indicates that 2 spaces are unreachable.
- 16 times *space 19* was the most visited space and 2 times *space 8*. Both of these spaces are central in the barn, so this is a realistic outcome. This statistic also indicates that all test cases produced similar results (with different numbers of entities and events) because the entities prefer the same spaces in each test.

- Using SMA *smoothing* instead of EMA results in slightly more entities being generated (not counting possible entities staying only in the outside space) and more movement regarding spaces and distance.

We conclude, that this experiment successfully proved that the outcomes of the test cases with different settings reflect what we would expect from them. This also means that there are no "bad" or "good" settings, rather they should be adjusted specifically to the use case.

Test Case	Total Locations	Distinct Entities	Covered Spaces	Most Visited Space	Total Spaces Moved	Average Spaces Moved per Entity	Average Distance Moved per Entity (in KM)
1	4648	10	16	19	3714	371.4	2.42
2	1762	10	16	19	1449	144.9	0.97
3	5685	40	16	19	4746	118.65	0.81
4	2598	45	16	8	2317	51.49	0.35
5	4275	45	16	19	3639	80.87	0.55
6	4190	45	16	19	3576	79.47	0.54
7	4398	45	16	19	3748	83.29	0.57
8	4468	45	16	19	3802	84.49	0.57
9	4543	45	16	19	3870	86	0.58
10	4605	9	16	19	3668	407.56	2.61
11	1712	9	16	8	1419	157.67	1.06
12	5664	42	16	19	4747	113.02	0.76
13	2781	47	16	19	2402	51.11	0.34
14	4582	47	16	19	3841	81.72	0.55
15	4588	47	16	19	3832	81.53	0.55
16	4866	47	16	19	4070	86.6	0.58
17	4909	47	16	19	4087	86.96	0.58
18	4875	47	16	19	4060	86.38	0.58

Table 4: Results of Experiment 1.

6.2 Experiment 2: Simulate Ground Truth Data

To find out if it is possible to simulate any location data accurately, we will try to simulate the ground truth data from the cattle test bed which we earlier used for training. The goal is to create a simulation with similar characteristics as the ground truth data. To compare the characteristics, we will calculate the statistics like in the first experiment.

6.2.1 Ground Truth Data Analysis

To gain knowledge of which characteristics the ground truth data has, we have to analyse it first. We apply the same statistics as in the first experiment to the ground truth data. We also use only data from 26 April 2022. This is the result:

- Total locations: 33450
- Distinct entities: 8
- Covered spaces: 20
- Most visited space: 2
- Total spaces moved: 33421
- Average spaces moved per entity: 4177.63
- Average distance moved per entity (in KM): 19.57

Comparing this result to the test cases of experiment 1, this seems very surprising. The cattle seem to move much more than expected, at least judging from this data.

6.2.2 Simulation of Ground Truth Data

Next, we will try to create a simulation with similar statistics as the ground truth data. This requires us to evaluate which settings to provide to the simulator. It is evident, that the *number of events* needs to be very high to create as much movement as the ground truth data has. We decided on the following settings with steadily increasing events:

- *smooth*: SMA
- *people: generation*: all
- *people: number*: 10

- *events: generation:* all
- *events: number:* 30000, 60000, 90000

The results, shown in table 5, are not as expected. It seems to be, that no matter how many events we generate, the entities do not move around more and more. The barrier seems to be somewhere around 550 events or 3.5 KM per entity. A logical explanation for this can be found in the internal logic of Scenario-Learning. One of the fixed settings we used is *time-thresh*, which determines a minimum duration to realize an event. This is specified by minutes in an integer value. We set it to the minimum, which is one minute. This limit prevents entities from "participating" in infinite events. In our case, this results in the thresholds we see in the results because the entities cannot participate in more events.

So, in the current version of SmartSPEC, it is impossible to generate a simulation with similar characteristics as the ground truth data. We would have to edit the source code of SmartSPEC, in particular the logic for the *time-thresh* parameter, to achieve this, which could be explored in future work.

Number of Events	Total Locations	Distinct Entities	Covered Spaces	Most Visited Space	Total Spaces Moved	Average Spaces Moved per Entity	Average Distance Moved per Entity (in KM)
30000	5934	8	16	2	4380	547.50	3.45
60000	6049	8	16	19	4442	555.25	3.46
90000	7410	10	16	2	5460	546	3.43

Table 5: Results of Experiment 2.

6.3 Requirements Fulfillment Check

In this section, we evaluate if the requirements defined in chapter 3 are fulfilled by the initial prototype of the simulator. We will reference the requirements by their *ID*. Confer the tables 1 and 2 for the respective description.

Fulfilled

- **REQ1**

We did not differentiate between barn and pasture scenarios for the implementation, as they are not any different from a simulation point of view. In our experiments, we have shown that the simulator works for the barn, which does not mean the pasture would require a different implementation.

- **REQ3**

The configuration for Scenario-Generation includes settings to adjust the amount of entities, which are in our case the animals, to generate. The number of sensors can be adjusted by defining more sensors in the *sensors* setting. The experiments have shown that sometimes an entity is totally omitted from the result, leading to marginally less entities generated than specified in the setting.

- **REQ4**

Experiment 1 has shown that the simulation resembles realistic movements of herd animals. There are central points which frequently passed and times when the animal does not move at all for a while. Also the average moved distance is realistic for herd animals.

- **REQ6**

By configuring the spaces, sensors and SmartSPEC settings in a way which resembles the desired use case, the simulator demonstrates the functionality of the WeideInsight application well. Experiment 1 showed that the results represent the settings accurately, which makes it possible to adjust the simulator for various demonstration purposes.

- **REQ7**

The Web UI we designed includes all possibilities the simulator has to offer in an intuitive way.

- **REQ13**

The backend was designed using a microservice architecture and the frontend was split into small components. Both approaches ensure that it is possible to extend the software easily.

- **REQ14**

The usage guides for frontend and backend make it clear that the software is easy to install and run. Docker helped us making the software easily installable and shippable.

Partly Fulfilled

- **REQ2**

It is possible to define the layout of the farm by configuring the spaces using the *Spaces* setting and entering it in a JSON format. In the future, the map could be advanced to configure the spaces intuitively.

- **REQ5**

The simulator can be easily integrated into WeideInsight, but an additional SimTrack component, which transforms the simulator locations to data which is compatible with the IPS, needs to be implemented to achieve this.

- **REQ8**

Historic data is currently provided by directly storing it in the respective database table *measurement*. The connection to the Track API is not yet implemented.

- **REQ15**

Docker secures the host system from malicious processes running in the docker containers. If the simulator was infiltrated it could not access the host system and would only affect the simulator application. But in the initial prototype there are many unhandled cases, which could be further secured.

Not (Yet) Fulfilled

- **REQ9**

This would hardly be possible using SmartSPEC, so an additional simulation method has to be integrated.

- **REQ10**

Only possible with additional simulation methods.

- **REQ11**

The integration into WeideInsight needs to be completed before this is possible.

- **REQ12**

This would require synchronisation between the simulator and WeideInsight in both directions. Future work needs to explore the possibilities here further.

7 Conclusion

As a final point of this thesis we will provide a short summary of the discussed topic and the results. After that, we will list down what could be done in the future regarding the implementation of the simulator and in the general topic of animal location simulation.

7.1 Summary

This thesis provided an overview of all research and development steps to create a simulator for location-based agriculture scenarios. We first discussed what motivated us to put work into this topic and why it is a necessary advancement for modern agriculture. To familiarise ourselves with the topic we looked into research done in the fields of simulation and smart spaces. This, and input from the stakeholders helped us listing down the requirements for the simulator in the scope of the WeideInsight project. Continuing on, we defined an architecture for both backend and frontend, to then implement it afterwards. By applying modern concepts, like microservice architectures, we created intuitive and customizable software. The evaluation of it showed that the simulator creates realistic simulations with limitations in the movement frequency. Nevertheless, the research question "How to develop a simulator for location models used in modern agriculture?" was answered successfully by the detailed step-by-step development process described throughout the thesis.

7.2 Future Work

Our implementation provided an initial prototype for the simulator with still limited capabilities, which do not fulfill all requirements just yet. We hinted towards several possible improvements, which are summarised here:

- Ability to configure farm layout, sensors and spaces using the map in the Web UI instead of JSON files
- Implementation of SimTrack component to make simulator compatible with WeideInsight infrastructure
- Implementation of connection to Track to pull learning data directly
- Addition of additional simulation methods to simulate location system and attributes of it
- Synchronization with IPS to be able to configure WeideInsight using the simulator
- User-Management including authentication and authorization

- Deployment of backend and frontend on a remote server
- Extensive evaluation of more settings to scope the limits of the simulator
- Adjustments to the SmartSPEC source code to overcome movement frequency limitation
- Implement simulation history to use previous simulation runs in the frontend

Apart from additional work on the simulator implementation, future research could explore the topic of animal location simulation further. Nearly all research available regarding location simulation deals with the simulation of humans only. As the herd behaviour of animals is very hard to express in mathematical form, more research has to be done to make this topic marketable.

References

- [1] M. Abdelaziz, *Designing Production-Grade and Large-Scale IoT Solutions*. Birmingham, UK: Packt Publishing, first ed., 2022.
- [2] K. Ashton, “That “Internet of Things” thing,” in *RFID Journal*, 2009.
- [3] A. Chio, D. Jiang, P. Gupta, G. Bouloukakis, R. Yus, S. Mehrotra, and N. Venkatasubramanian, “Smartspec: customizable smart space datasets via event-driven simulations,” in *PERCOM 2022: 20th International Conference on Pervasive Computing and Communications*, (Pisa, Italy), pp. 152–162, IEEE, Mar. 2022.
- [4] M. Sünkel, G. Elmamooz, M. Grawunder, P. T. Hoang, E. Rauch, L. Schmeling, S. Thurner, and D. Nicklas, “Resource-aware classification via model management enabled data stream optimization,” in *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 181–182, IEEE, 2022.
- [5] “Weideinsight.” <https://www.uni-bamberg.de/weideinsight/>, 2022. [Online; accessed 15-Feb-2023].
- [6] T. Ören, “The many facets of simulation through a collection of about 100 definitions,” in *SCS M&S Magazine*, vol. 2 (April), pp. 82–92, 2011.
- [7] T. Ören, B. P. Zeigler, and A. Tolk, *Body of Knowledge for Modeling and Simulation*. Cham, Switzerland: Springer Nature Switzerland AG, first ed., 2023.
- [8] J. Bruneau and C. Consel, “Diasim: A simulator for pervasive computing applications,” in *Software: Practice and Experience*, vol. 43 (8), pp. 82–92, 2011.
- [9] V. Kulkarni, N. Tagasovska, T. Vatter, and B. Garbinato, “Generative models for simulating mobility trajectories,” 2018.
- [10] A. Roy, R. Fablet, and S. L. Bertrand, “Using generative adversarial networks (gan) to simulate central-place foraging trajectories,” *Methods in Ecology and Evolution*, vol. 13, no. 6, pp. 1275–1287, 2022.
- [11] “A gentle introduction to generative adversarial networks (gans).” <https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/>, 2019. [Online; accessed 18-Feb-2023].
- [12] L. Pappalardo and F. Simini, “Data-driven generation of spatio-temporal routines in human mobility,” *Data Mining and Knowledge Discovery*, vol. 32, pp. 787–829, dec 2017.

-
- [13] S. Ha and J. Kwon, “Visual tracking enhancement by trajectory simulation based on hidden semi-markov model,” *Electronics Letters*, vol. 56, no. 2, pp. 85–87, 2020.
 - [14] S. Balandin and H. Waris, “Key properties in the development of smart spaces,” in *Universal Access in Human-Computer Interaction. Intelligent and Ubiquitous Interaction Environments* (C. Stephanidis, ed.), (Berlin, Heidelberg), pp. 3–12, Springer Berlin Heidelberg, 2009.
 - [15] S. Helal and S. Tarkoma, “Smart spaces [guest editors’ introduction],” *IEEE Pervasive Computing*, vol. 14, no. 02, pp. 22–23, 2015.
 - [16] “Github - andrewgchio/smartspec.” <https://github.com/andrewgchio/SmartSPEC/>, 2021. [Online; accessed 23-Mar-2023].
 - [17] U. of Bamberg Chair of Mobile Systems, “Vorhabenbeschreibung “WeideInsight”,” 2021.
 - [18] “mioty alliance.” <https://mioty-alliance.com/>, 2023. [Online; accessed 19-Feb-2023].
 - [19] “Safactory track.” <https://track.safactory.com/>, 2023. [Online; accessed 19-Feb-2023].
 - [20] F. Dalpiaz and A. Sturm, “Conceptualizing requirements using user stories and use cases: A controlled experiment,” in *Requirements Engineering: Foundation for Software Quality* (N. Madhavji, L. Pasquale, A. Ferrari, and S. Gnesi, eds.), (Cham), pp. 221–238, Springer International Publishing, 2020.
 - [21] F. Heath, *Managing Software Requirements the Agile Way: Bridge the Gap Between Software Requirements and Executable Specifications to Deliver Successful Projects*. Packt Publishing, 2020.
 - [22] F. Halili and E. Ramadani, “Web services: A comparison of soap and rest services,” *Modern Applied Science*, vol. 12, p. 175, 02 2018.
 - [23] K. Köhler, *Microservices mit Go*. Rheinwerk Verlag, first ed., 2021.
 - [24] “The go programming language.” <https://go.dev/>, 2023. [Online; accessed 05-Mar-2023].
 - [25] C. Ware, *Information Visualization: Perception for Design*, vol. 4 of *Interactive Technologies*. Elsevier Science, 2019.
 - [26] M. Schütze, P. Sachse, and A. Römer, “Support value of sketching in the design process,” *Research in Engineering Design*, vol. 14, no. 2, pp. 89–97, 2003.

-
- [27] “Excalidraw.” <https://excalidraw.com/>, 2023. [Online; accessed 17-Mar-2023].
 - [28] M.-J. Kraak, “The space-time cube revisited from a geovisualization perspective,” *Proc 21st Int Cartogr Conf*, 07 2008.
 - [29] “Docker: Accelerated, containerized application development.” <https://www.docker.com/>, 2023. [Online; accessed 05-Mar-2023].
 - [30] “Docker compose overview.” <https://docs.docker.com/compose/>, 2023. [Online; accessed 05-Mar-2023].
 - [31] A. Shuiskov, *Microservices with Go*. Birmingham, UK: Packt Publishing, first ed., 2022.
 - [32] T. Collings, “Controller-service-repository.” <https://tom-collings.medium.com/controller-service-repository-16e29a4684e5>, 2021. [Online; accessed 18-Mar-2023].
 - [33] J. Paul, “Top 5 programming languages for web development in 2023.” <https://medium.com/javarevisited/top-5-programming-languages-for-web-development-in-2021-f6fd4f564eb6> 2021. [Online; accessed 19-Mar-2023].
 - [34] L. S. Vailshery, “Most used web frameworks among developers worldwide, as of 2022.” <https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/>, 2022. [Online; accessed 19-Mar-2023].
 - [35] “React the library for web and native user interfaces.” <https://react.dev/>, 2023. [Online; accessed 19-Mar-2023].
 - [36] S. Blackheath and A. Jones, *Functional Reactive Programming*. New York, USA: Manning Publications, first ed., 2016.
 - [37] “Reactbootstrap reactbootstrap documentation.” <https://react-bootstrap.github.io/>, 2023. [Online; accessed 19-Mar-2023].
 - [38] “Github axios/axios: Promise based http client for the browser and node.js.” <https://github.com/axios/axios>, 2023. [Online; accessed 19-Mar-2023].
 - [39] F. Klinker, “Exponential moving average versus moving exponential average,” *Mathematische Semesterberichte*, vol. 58, pp. 97–107, dec 2010.

A Sources

Source	Description
GitLab Repository	https://gitlab.rz.uni-bamberg.de/mobi/theses/2022-ma-paul-pongartz
Thesis Report	In Repository -> /report/final-thesis-report.pdf
Source Code	In Repository -> folder /code
Presentations	In Repository -> folder /presentations

B Experiment 1 - Test Case 1: Settings

Sensors

```
[
  {
    "id": 1,
    "metasensor-id": 1,
    "coverage": [1],
    "coordinates": [3,3,1]
  },
  {
    "id": 2,
    "metasensor-id": 1,
    "coverage": [2],
    "coordinates": [3,2,1]
  },
  {
    "id": 3,
    "metasensor-id": 1,
    "coverage": [3],
    "coordinates": [3,1,1]
  },
  {
    "id": 4,
    "metasensor-id": 1,
    "coverage": [4],
    "coordinates": [3,0,1]
  },
  {
    "id": 5,
    "metasensor-id": 1,
    "coverage": [5],
    "coordinates": [2,0,1]
  },
  {
    "id": 6,
    "metasensor-id": 1,
    "coverage": [6],
    "coordinates": [2,1,1]
  },
  {
    "id": 7,
    "metasensor-id": 1,
    "coverage": [7],
    "coordinates": [2,2,1]
  },
]
```

```
{
  "id": 8,
  "metasensor-id": 1,
  "coverage": [8],
  "coordinates": [2,4,1]
},
{
  "id": 9,
  "metasensor-id": 1,
  "coverage": [9],
  "coordinates": [1,3,1]
},
{
  "id": 10,
  "metasensor-id": 1,
  "coverage": [10],
  "coordinates": [1,2,1]
},
{
  "id": 11,
  "metasensor-id": 1,
  "coverage": [11],
  "coordinates": [1,1,1]
},
{
  "id": 12,
  "metasensor-id": 1,
  "coverage": [12],
  "coordinates": [1,0,1]
},
{
  "id": 13,
  "metasensor-id": 1,
  "coverage": [13],
  "coordinates": [0,0,1]
},
{
  "id": 14,
  "metasensor-id": 1,
  "coverage": [14],
  "coordinates": [0,1,1]
},
{
  "id": 15,
  "metasensor-id": 1,
  "coverage": [15],
  "coordinates": [0,2,1]
```

```

    },
    {
      "id": 16,
      "metasensor-id": 1,
      "coverage": [16],
      "coordinates": [0,3,1]
    },
    {
      "id": 17,
      "metasensor-id": 1,
      "coverage": [17],
      "coordinates": [0,4,1]
    },
    {
      "id": 18,
      "metasensor-id": 1,
      "coverage": [18],
      "coordinates": [1,4,1]
    },
    {
      "id": 19,
      "metasensor-id": 1,
      "coverage": [19],
      "coordinates": [2,3,1]
    }
  ]

```

Spaces

```

[
  {
    "id": 0,
    "description": "outside",
    "capacity": -1,
    "neighbors": [
      8
    ],
    "coordinates": [6,1,1],
    "geoCoordinates": [49.68181426283488,12.198898781537865,1]
  },
  {
    "id": 1,
    "capacity": 10,
    "neighbors": [
      2, 19
    ],
    "coordinates": [3,3,1],
    "geoCoordinates": [49.68168614140271,12.199246531983448,1]
  }
]

```

```
},
{
  "id": 2,
  "capacity": 10,
  "neighbors": [
    7, 3, 1
  ],
  "coordinates": [3,2,1],
  "geoCoordinates": [49.68163864109388,12.199289060635687,1]
},
{
  "id": 3,
  "capacity": 10,
  "neighbors": [
    4, 6, 2
  ],
  "coordinates": [3,1,1],
  "geoCoordinates": [49.68158996138931,12.19933106498558,1]
},
{
  "id": 4,
  "capacity": 10,
  "neighbors": [
    5, 3
  ],
  "coordinates": [3,0,1],
  "geoCoordinates": [49.6815416168796,12.199374698510486,1]
},
{
  "id": 5,
  "capacity": 10,
  "neighbors": [
    6, 12, 4
  ],
  "coordinates": [2,0,1],
  "geoCoordinates": [49.681507303985796,12.19927908068362,1]
},
{
  "id": 6,
  "capacity": 10,
  "neighbors": [
    11, 5, 7, 3
  ],
  "coordinates": [2,1,1],
  "geoCoordinates": [49.68155539409804,12.199236814182555,1]
},
{
  "id": 7,
```

```
"id": 7,
"capacity": 10,
"neighbors": [
  10, 6, 2, 19
],
"coordinates": [2,2,1],
"geoCoordinates": [49.681604348471495,12.199194191858536,1]
},
{
  "id": 8,
  "capacity": 10,
  "neighbors": [
    18, 19
  ],
  "coordinates": [2,4,1],
  "geoCoordinates": [49.681660897440366,12.199142826903778,1]
},
{
  "id": 9,
  "capacity": 10,
  "neighbors": [
    16, 10, 18, 19
  ],
  "coordinates": [1,3,1],
  "geoCoordinates": [49.681617047119886,12.199057964795283,1]
},
{
  "id": 10,
  "capacity": 10,
  "neighbors": [
    15, 11, 9, 7
  ],
  "coordinates": [1,2,1],
  "geoCoordinates": [49.68156855583595,12.19910049056047,1]
},
{
  "id": 11,
  "capacity": 10,
  "neighbors": [
    14, 12, 10, 6
  ],
  "coordinates": [1,1,1],
  "geoCoordinates": [49.681521095937654,12.19914174430126,1]
},
{
  "id": 12,
  "capacity": 10,
```

```
    "neighbors": [
      13, 11, 5
    ],
    "coordinates": [1,0,1],
    "geoCoordinates": [49.681472604558024,12.199184270066445,1]
  },
  {
    "id": 13,
    "capacity": 10,
    "neighbors": [
      12, 14
    ],
    "coordinates": [0,0,1],
    "geoCoordinates": [49.681454888552196,12.199139437638001,1]
  },
  {
    "id": 14,
    "capacity": 10,
    "neighbors": [
      13, 11, 15
    ],
    "coordinates": [0,1,1],
    "geoCoordinates": [49.68150332868538,12.199099995195867,1]
  },
  {
    "id": 15,
    "capacity": 10,
    "neighbors": [
      14, 10, 16
    ],
    "coordinates": [0,2,1],
    "geoCoordinates": [49.68155265441732,12.199056744575499,1]
  },
  {
    "id": 16,
    "capacity": 10,
    "neighbors": [
      9, 17, 15
    ],
    "coordinates": [0,3,1],
    "geoCoordinates": [49.681599431109674,12.199011649936438,1]
  },
  {
    "id": 17,
    "capacity": 10,
    "neighbors": [
      16, 18
```

```

    ],
    "coordinates": [0,4,1],
    "geoCoordinates": [49.68164737378305,12.198968064039947,1]
  },
  {
    "id": 18,
    "capacity": 10,
    "neighbors": [
      17, 9, 8
    ],
    "coordinates": [1,4,1],
    "geoCoordinates": [49.68166940624579,12.199029922485352,1]
  },
  {
    "id": 19,
    "capacity": 10,
    "neighbors": [
      9, 7, 8, 2
    ],
    "coordinates": [2,3,1],
    "geoCoordinates": [49.681637855756065,12.199179120361807,1]
  }
]

```

Metasensors

```

[
  {
    "id" : 1,
    "description" : "Bluetooth Beacon"
  }
]

```

Scenario-Learning Configuration

```

[learners]
start      = 2022-04-26
end        = 2022-04-28
unit       = 1
validity   = 1
smooth     = EMA
window     = 10
time-thresh = 1
occ-thresh = 1

```

Scenario-Generation Configuration

```

[people]
number = 10

```

```
generation = all
```

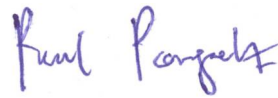
```
[events]  
number = 3000  
generation = all
```

```
[synthetic-data-generator]  
start = 2022-04-26  
end    = 2022-04-26
```

Versicherung

Ich erkläre hiermit gemäß §9 Abs. 12 APO, dass ich die vorstehende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren erkläre ich, dass die digitale Fassung der gedruckten Ausfertigung der Masterarbeit ausnahmslos in Inhalt und Wortlaut entspricht und zur Kenntnis genommen wurde, dass diese digitale Fassung einer durch Software unterstützten, anonymisierten Prüfung auf Plagiate unterzogen werden kann.

Mittwoch, den 29.03.2023



Paul Pongratz