**BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA**
**FACULTY OF MATHEMATICS AND COMPUTER SCIENCE**
**SPECIALIZATION Computer Science**

# DIPLOMA THESIS

# FlexiTime - A deep dive into the developement of an Automatic Schedule Generator

**Supervisor**
**Lect. PhD. Mursa Bogdan-Eduard-Mădălin**

*Author*
*Octavian-Mihail Gheorghe*

2025

**UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA**
**FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ**
**SPECIALIZAREA INFORMATICĂ**

# LUCRARE DE LICENŢĂ

# FlexiTime – O analiză aprofundată a dezvoltării unui generator automat de orare

**Conducător științific**
**Lect. Dr. Mursa Bogdan-Eduard-Mădălin**

*Absolvent*
*Octavian-Mihail Gheorghe*

2025

ABSTRACT

In an age of increasing complexity and time pressure, effective personal time management has become a persistent challenge. Traditional digital calendars like Google Calendar or Microsoft Outlook offer scheduling assistance but depend heavily on manual input and lack intelligent, adaptive behavior. This thesis addresses the problem of automatic personalized schedule generation by developing a system that builds and dynamically maintains user-centric schedules based on defined activities, constraints, and preferences.

To meet this challenge, the work adopts a heuristic method grounded in Squeaky Wheel Optimization (SWO), a fast and flexible iterative search framework that prioritizes difficult-to-schedule tasks and incrementally adapts schedules based on user behavior. The system supports both hard constraints (e.g., deadlines, availability windows) and soft preferences (e.g., preferred times, task spacing).

The thesis includes a comprehensive theoretical analysis of scheduling methodologies, comparing heuristic constraint-based and machine learning approaches. SWO was selected for its balance between computational efficiency, freedom in expression by numerous user choices, and real time adaptability. The resulting application features a user interface for task entry and schedule generation, supported by a robust backend capable of functionalities such as handling interruptible tasks, and priority re-evaluation.

Experimental results confirm that the proposed system effectively generates personalized and adaptable plans with minimal user input, offering a practical solution for intelligent personal time management.

*During the preparation of this work, we used OpenAi tools to find and summarize some research papers and to help in troubleshooting any issues encountered during the developement of the application. All AI-generated content, suggestions, and ideas were reviewed, adapted, and verified. The use of AI tools supported the development process and decision-making, but did not replace our original contributions or critical judgment. We take full responsibility for the accuracy and integrity of the content presented in this thesis.*

*This work is the result of my own activity, and I confirm I have neither given, nor received unauthorized assistance for it."*

# Contents

# Chapter 1

# Introduction

## 1.1 Context Motivation

Personal time management is a fundamental challenge faced by individuals in today's increasingly complex and dynamic environments. Although traditional electronic calendars such as Google Calendar or Microsoft Outlook offer functions such as event reminders, conflict detection, and meeting scheduling, they still rely heavily on manual user input, which is the greatest challenge, as people have less and less time to fulfill their wants and needs [RYS10]. These systems provide little support for intelligent, automatic generation of personalized schedules that adapt to user preferences and dynamically changing contexts.

Maintaining management of daily activities involves more than just filling in time slots. It requires balancing multiple constraints, sometimes even conflicting ones, such as task deadlines, task dependencies, preferred working periods, and the varying levels of flexibility tasks may allow (for example splitting a time-dense activity across multiple intervals and, if necessary, even multiple locations). As people's lives become busier, there is a growing need for systems capable of autonomously proposing meaningful, adaptable, and preference-aware schedules [Ref07, AR16].

Developing such an intelligent scheduling system poses several significant challenges: modeling human tasks realistically, efficiently resolving complex constraints of both temporal and spatial origin, and constantly taking into consideration all of the user's preferences.

## 1.2 Project Goal

This thesis addresses the problem of automatic personalized schedule generation. The objective is to design and implement an application that, based on user-defined

activities and preferences, can automatically generate an optimized schedule for the user, while respecting both hard constraints (such as deadlines and task order) and soft preferences (as are preferred times for certain activities).

To achieve this, the thesis adopts a heuristic approach based on the Squeaky Wheel Optimization (*SWO*) framework, an iterative search method known and often utilized for its effectiveness in solving large scale scheduling problems under time constraints [Ref07]. SWO provides a flexible structure to prioritize tasks which are more difficult to schedule and to quickly generate timetables that prioritize the user's satisfaction, even if at the detriment of a more globally optimal strategy.

The strength of the Squeaky Wheel Optimization approach lies in its ability to dynamically react to scheduling challenges by prioritizing "troublemaker" tasks, those that are hardest to schedule due to constraint violations or resource contention [RA11]. This adaptive mechanism makes SWO particularly suitable for personal scheduling, where tasks often compete for limited time slots and user preferences are complex and nuanced. Unlike traditional optimization methods that aim for a single globally optimal solution, SWO embraces heuristic flexibility and iterative refinement to produce high quality, user centric schedules.

By integrating preference learning and the potential for generating alternative plans which compete with one another in regards to their utility metrics [AR16, RYS10], the system can evolve in response to the user's requirements. This makes it a robust, intelligent foundation for building practical, personalized scheduling systems.

## 1.3 Structure of the Thesis

Since the focus of the thesis is creating an application, alongside explaining the design of the algorithm in a theoretical foundation part, we will also provide an experimental/practical implementation one. Therefore, the structure provided is:

**Chapter 2** defines the scheduling problem formally, describing the types of tasks, constraints, and preferences the system needs to manage;

**Chapter 3** presents a detailed state of the art review, analyzing different approaches to automatic scheduling, tackling the 2 main problem solutions, that of machine learning and heuristic methods [GKAU24, SH25], while also discussing their intricacies and their subcategories, with a focus on the rationale for adopting a heuristic approach [Sil04];

**Chapter 4** provides an in-depth theoretical description of the selected algorithm,

Squeaky Wheel Optimization, and its adaptations for the purposes of personalized scheduling [Sat20];

**Chapter 5** introduces the proposed solution architecture, discussing design decisions for both backend algorithms and user facing functionalities.

**Chapter 6** covers experimental validation, including evaluations of scheduling quality, adaptability, and computational efficiency;

**Chapter 7** describes the testing approach and methodology used to verify application correctness.

**Chapter 8** describes the final application from a user's perspective, including prerequisites, and a usage manual via screenshots;

**Chapter 9** concludes the thesis by summarizing the contributions, discussing limitations, and suggesting potential future improvements;

Through this structured development, the thesis aims to contribute both a functional scheduling application and a deeper understanding of heuristic methods for personal time management automation.

# Chapter 2

# Problem Definition

## 2.1 Defining the Problem

Efficiently managing one's personal schedule involves balancing multiple activities, constraints, and user preferences, often under dynamic and uncertain conditions. Traditional calendar systems mainly provide users the ability to manually place tasks into empty slots, detecting conflicts, or setting reminders. However, they lack the algorithms needed to autonomously generate and adapt user specific schedules based on personal goals and habits [RA11] and this inevitably results in schedules being dropped and forgotten continuously.

In this thesis, we shall tackle the problem of automatic personalized schedule generation, defined as such: **Given a set of user defined activities, constraints and preferences, alongside the locations of the activities, generate a feasible schedule that satisfies all of the hard constraints and aims to align with the user's soft preferences**.

The scheduling system must:

- Respect hard constraints: physical feasibility such as non-overlapping activities, precedence relations (for example, task A must happen before task B), and various time windows for all Activities [Ref07];
- Optimize soft preferences: user desired time slots and duration preferences, preferred task orderings, or proximity [RYS10];
- Handle activity splitting: divide interruptible tasks into valid sub parts that respect duration and spacing constraints while remaining feasible [Ref07];
- Maximize scheduling utility: iteratively construct and evaluate schedules, terminating when utility/difficulty converges [AR16];

## 2.2 Understanding User's Preferences

Each user has the ability to define activities that have the following characteristics, based on the models and expressiveness found in SelfPlanner [RA11] and related work [RYS10]:

- **Duration:** Minimum and maximum possible durations for the task are mandatory. They are user-defined and represent the lower and upper bounds for the total duration of all the parts that an activity may be split into. We denote these bounds as $\texttt{dur}_{\text{min}}$ and $\texttt{dur}_{\text{max}}$, such that:

$$\texttt{dur}_{\text{min}} \leq \sum_{i=1}^{p} d_i \leq \texttt{dur}_{\text{max}}$$

  where $d_i$ is the duration of part $i$, and $p$ represents the number of parts;

- **Time Windows:** Each activity has a set of available temporal intervals:

$$F = \{[a_1, b_1] \cup [a_2, b_2] \cup [a_3, b_3] \cup \cdots \cup [a_F, b_F]\}$$

  during which it is allowed to be scheduled. These are user-defined and must satisfy $a_i < b_i$ for all $i$. Every activity part must be scheduled without breaking either the lower or the upper bound of any interval;

- **Locations:** Each activity is associated with a set of valid locations

$$L = \{\ell_1, \ell_2, \ldots, \ell_m\}$$

  where each $\ell_i$ is defined as a tuple $(\texttt{name}, x, y)$, representing its identifier and spatial coordinates. The scheduler uses these coordinates[TEK25] to compute travel times between consecutive locations and avoid infeasible transitions. All of the data regarding the locations and the distance between them is located inside a distance matrix initialized before the start of the SWO algorithm;

- **The ability to be split into Parts:** Activities may be divided into $p$ parts, each represented by a tuple $(t_i, dur_i, \ell_i)$, where $t_i$ represents the start time, $dur_i$ is the duration, and $\ell_i \in L$ is the assigned location. The algorithm determines the optimal value of $p$ and the placement of each part according to constraints and utility, which we will get into later on;

- **Minimum and Maximum Part Duration:** Users may define preferred duration bounds for each individual part of an interruptible activity, using $s_{\text{min}}$ and

$s_{\text{max}}$. These bounds must be respected as:

$$s_{\text{min}} \leq dur_i \leq s_{\text{max}}, \quad \forall i \in \{1, \ldots, p\}$$

If unspecified, the activity is considered non-interruptible and treated as a single block (i.e., $p = 1$);

- **Minimum and Maximum Part Distance:** To specify how far apart different parts of the same activity should be scheduled, users can define spacing bounds $d_{\text{min}}$ and $d_{\text{max}}$. These enforce:

$$d_{\text{min}} \leq t_{i+1}^{\text{start}} - t_i^{\text{end}} \leq d_{\text{max}}, \quad \forall i \in \{1, \ldots, p-1\}$$

where $t_i^{\text{end}} = t_i^{\text{start}} + dur_i$.

- Constraints Between Activities:
  - **Minimum and Maximum Part Distance Constraints:** This constraint's purpose is to allow the users to experiment with the schedules they can get without yet modifying the domains of the activity generation guidelines. These constraints are the only user-defined unary constraints;
  - **Proximity Constraints:** This category also encapsulates 2 constraints. They are similar to the minimum/maximum part distance constraint, but they no longer involve the parts of only 1 activity, but 2. This allows for interleaved sectioning of the respective activities while also making sure that all of their parts are close/far enough to/from each other (For example, "minimum 30 minutes break after gym before dinner");
  - **Ordering Constraints:** The last of the 3 binary constraints. They basically allow the user to have even more control over the order in which activity parts are placed in relation to each other (An example for this constraint's use would be: "buy ingredients" before "cook dinner");
- Unary Preferences for Activities:
  - **Activity Total Duration Preference:** This preference allows the user to indicate a desirable duration interval within the hard constraints of minimum and maximum duration. For instance, a user might define a task with $dur_{Min}$ = `60` and $\texttt{d}ur_{Max} = 180$ (minutes), but express a preference that the task ideally lasts `90 - 120` minutes. While the scheduler can assign any value between 60 and 180, schedules closer to the preferred value are ranked more favorably in utility;

  - **Early/Late Activity Placement Preference:** This preference expresses the user's inclination for scheduling a task earlier or later within its allowed time windows. For example, if a user has a time window of 8:00 – 12:00, but prefers to perform the activity earlier in that window (as example, between 8:00 and 9:00 being the most optimal), then the algorithm will assign more utility to schedules that place the activity near the beginning of its domain. The preference can be either ascending (late is better) or descending (early is better);
  - **Minimum/Maximum Part Duration Preference:** Analogous to the part duration constraint, but softer. This lets the user indicate a preferred range for how long each part of an interruptible activity should last. For instance, even if $s_{min} = 30$ and $s_{max} = 90$, the user may prefer each part to be around 60 minutes. This preference guides the schedule generator to favor such configurations without disqualifying other feasible options.

- Binary Preferences between Activities:

  - **Implication Preference:** This binary preference allows the user to express a logical dependency where one activity should ideally occur only if another is scheduled. For example, a user might state a preference like: "If I attend a gym session, I'd like to also schedule a sauna session afterward." Unlike constraints, preferences of this kind do not enforce hard behavior but increase the schedule's utility when the implication holds.
  - **Minimum/Maximum Activities Distance Preference:** This functions similarly to proximity constraints but does not disqualify schedules that do not comply. Instead, it encourages the scheduler to place two different activities closer together or farther apart in time, depending on the user's preference. For example, if the user prefers at least a 30-minute gap between "Lunch" and "Meeting", schedules respecting that interval will receive higher utility, while others remain valid but less favored.
  - **Ordering Preference:** This is a soft counterpart to the ordering constraint. The user may indicate that it is preferable (but not mandatory) for one activity to occur before another. For instance, "prefer doing homework before doing chores" means schedules satisfying this order get a utility boost, but the opposite order is still allowed.

The separation of hard constraints and soft preferences, and the use of structured preference models (as is early/late preferences, splitting, logical implication) are fundamental in such scheduling problems [RYS10, AR16].

## 2.3 Requirements and Challenges

Designing a fully personalized, adaptive scheduler presents several significant challenges:

### 2.3.1 Conflicting Preferences

Users may express conflicting soft preferences (for example, wanting a task both early in the day and after another task that itself is preferred early). The scheduling engine must have a system that prioritizes intelligently balancing different degrees of satisfaction across various combination of preferences provided by the user [AR16]. Usually, binary preferences will be preferred as the goal of the application is also to provide schedules filled with as many activities as possible. Thus any preference regarding the placement of 2 activities will matter more to the overall utility.

### 2.3.2 Activity Splitting

Interruptible tasks require logic that splits them into smaller valid chunks while satisfying minimum and maximum chunk sizes, and respecting minimal distances between parts [Ref07]. The algorithm will try to fit in as much of activity's max duration in the resulting schedule while taking into account all of the preferences laid out by the user. Not only that, once it reaches the minimum amount of duration required by the user for the scheduling of the activity to be considered finished, the algorithm shall take care in not generating any more activity parts if it turns out to be a huge detriment to the scheduling of all the remaining activities.

### 2.3.3 Precedence and Dependencies

Tasks may have both ordering or dependency relationships, and the scheduler must ensure these are honored even under dynamic changes [RYS10, Ref07]. Constraints are constantly checked against the currently built schedule to ensure that none are ever broken and that utility (which comes from both constraints AND preferences) is maximized.

### 2.3.4 Computational Efficiency

Given the complexity of personal preferences and constraints, optimal solutions are often computationally expensive. However, users expect quick response times.

Therefore, the solution must be heuristic: good enough, quickly, rather than necessarily optimal [RA11].

### 2.3.5 Ease of Use and Understanding

This application also posess the problem of accesibility. The generation of schedules and their result must not only be feasible but also easy for the user to understand and follow. Overly complex or counterintuitive plans would diminish user trust and satisfaction, hence why the algorithm provides the use of binary constraints. Ensuring the ease of access for everyone means also providing an intuitive method for them to setup activity generation guidelines, with their respective locations, temporal intervals, constraints and preferences.

# Chapter 3

# State of the art

## 3.1 Traditional Approaches

Traditional digital calendars primarily assist users by allowing manual scheduling of tasks, providing notifications, and detecting basic conflicts. While all of these things are valuable and even necessary to a calendar application, these systems fundamentally rely on human input and do not autonomously generate schedules based on complex preferences or constraints [Ref07]. This inherent limitation inspired researchers to explore automated methods for personal schedule generation, using diverse computational paradigms.

Over the years, around 2 major families of approaches have emerged:

- Heuristic Search methods;
- Machine Learning-based methods;

with each of them having their own subcategories and methods to consider.

Each methodology presents distinct advantages and drawbacks when applied to the problem of personal activity scheduling.

## 3.2 Heuristic Approaches

Heuristic algorithms[Sil04] rely on informed search strategies and rule of thumb techniques to guide the scheduling process. They typically follow an iterative procedure where activities are evaluated and ordered according to certain priority measures, a solution is generated based on those priorities, and then refined. The general pipeline often includes the following steps:

- **Priority Assignment:** Activities are assigned initial priorities, often based on utility, difficulty of placement, or constraint tightness.

- **Greedy Construction:** A feasible schedule is generated by placing activities in order of priority using a greedy strategy.
- **Troublemaker Identification:** Activities that cause constraint violations or reduce overall schedule quality are identified and reprioritized.
- **Iterative Refinement:** The process repeats, adjusting priorities and regenerating schedules until a termination condition is met, usually utility/difficulty convergence or a time/resource limit.

Heuristic approaches can be broadly categorized into several families, including rule-based scheduling, greedy constructive heuristics, and metaheuristics such as Genetic Algorithms (GA) [REK23], Ant Colony Optimization (ACO)[GKAU24], and Particle Swarm Optimization (PSO). Among these, the *Squeaky Wheel Optimization* [JC99] algorithm has gained special attention for personal task scheduling due to its adaptability and simplicity.

### 3.2.1 Squeaky Wheel Optimization

Squeaky Wheel Optimization (SWO), introduced by Joslin and Clements (1999), operates on a construct-analyze-prioritize loop. A greedy algorithm first constructs a solution based on current activity priorities. The solution is then analyzed to find problematic components ("squeaky wheels"), whose priorities are increased to improve their placement in subsequent iterations [JC99]. This loop continues until the utility of the resulting schedule stabilizes.

In Refanidis and Yorke-Smith's work on *Managing Personal Tasks with Time Constraints and Preferences*[Ref07], SWO was adapted for individual task scheduling by introducing the idea of adding more variables such as the time taken in traversing the distance between activities. Their system, SELFPLANNER, supports:

- Preemptive (interruptible) and non-preemptive tasks
- Multiple time intervals (domains) per activity
- Spatial reasoning between alternative locations
- Temporal and ordering constraints (e.g., "task A before task B")
- User-defined preferences for duration and time location

The goal was to maximize total schedule utility by combining multiple utility sources (task inclusion, temporal alignment, proximity, and duration satisfaction). Each schedule is evaluated as a Constraint Optimization Problem (COP), where constraint satisfaction is balanced against soft preference fulfillment [RYS10].

### 3.2.2 Constraint-Based vs. Utility-Based Heuristics

Initially, SWO in SELFPLANNER was driven by constraint satisfaction: activities violating constraints were prioritized for improvement. However, an important intermediate work deepened the constraint model and integrated stronger utility evaluation [RYS10]. This enhanced version fully supported splitting activities into parts, scheduling them across time and locations, and iteratively maximizing utility until convergence. In fact this specific paper represented the first time the term of "utility" was defined in the context of personal task scheduling.

Later research shifted even more toward utility-driven planning [AR16], where schedules were not only ranked primarily on user satisfaction rather than feasibility alone, but also even more reliant on the user's preferences, since they were taken into consideration inside the scheduling problem by considering his/her former choices in older schedule generations. This allowed small soft-preference trade-offs to achieve higher overall utility.

Key improvements in this stage included:

- **Alternative Plan Generation:** Generating multiple near-optimal schedules that are qualitatively different.
- **Online Preference Learning:** Monitoring user choices to adapt the preference model over time.

### 3.2.3 Other Heuristic Families

While SWO is particularly well-suited to individual scheduling, other metaheuristic approaches have also been explored:

- **Genetic Algorithms (GA):** These simulate evolution by recombining candidate schedules. They explore large spaces but typically need more tuning. [REK23]
- **Ant Colony Optimization (ACO):** Inspired by ant foraging, these build schedules using pheromone-guided probabilistic placement. [GKAU24]
- **Particle Swarm Optimization (PSO):** Based on flocking behavior, PSO adjusts candidate schedules through social-inspired heuristics [GKAU24].

Although effective, these methods are often more complex or less transparent to users. In contrast, SWO offers intuitive logic, fast convergence, and interpretability.

### 3.2.4 Strengths and Weaknesses

The primary strengths of heuristic approaches, particularly SWO-based ones, include:

- **Speed and Scalability:** Quickly finds good solutions for large problems [JC99].
- **Flexibility:** Supports partial preferences during generation, coonstraints, and dynamic tasks.
- **User Alignment:** Easily integrates preference learning and alternative plan features [AR16].

However, these methods are not without limitations:

- **Suboptimality:** Heuristics aim for satisfactory solutions, not guaranteed global optimality.
- **Heuristic Sensitivity:** Poor prioritization heuristics can degrade performance.
- **Non-Determinism:** Outcomes may vary due to randomness, initial ordering or small road bumps that were not properly taken into account during developement.

Despite these drawbacks, for the context of this thesis, designing a personal, adaptive, and interactive scheduling system, heuristic methods offer a strong foundation. Their balance between computational performance and alignment with user needs makes them ideal for user-facing applications.

## 3.3 Machine Learning (ML) Approaches

In contrast to heuristic methods, machine learning (ML) approaches aim to learn scheduling behavior from data, adapting to user patterns and dynamic contexts. These systems can automatically adapt to user behavior, environment dynamics, and even noisy or incomplete inputs, which makes them highly appealing for personalized scheduling.[Sat20]

### 3.3.1 Supervised Learning and Preference Modeling

One promising line of work is to use supervised models to predict preferred event timings or sequences. A notable example is the Neural Event Scheduling Assistant (NESA), which employs BiLSTM, CNN, and highway layers to infer calendar preferences from over 593K events [KLC+18]. NESA can detect subtle scheduling habits without explicit rules.

These models excel at capturing such implicit scheduling tendencies and can tolerate unstructured data (for example, vague or misspelled event titles). However, they typically require large-scale, high-quality datasets for training and may struggle to generalize with sparse or cold-start users. Moreover, integrating hard constraints (for example, deadlines, dependencies) is not straightforward in purely predictive models.

### 3.3.2 Reinforcement Learning and Graph Neural Networks

A more recent trend applies reinforcement learning (RL) to scheduling. Platten et al. (2022) modeled personnel scheduling as a graph and trained a GNN policy via REINFORCE to allocate employees to shifts while respecting complex constraints. Their approach significantly outperformed non-learning baselines across varied test sets [PMGM22].

Their methodology includes:

- Representing each scheduling problem as a dynamic graph structure.
- Using an encoder-decoder GNN trained via the REINFORCE[PMGM22] algorithm, a policy-gradient RL method.
- Modeling the agent's actions as probabilistic assignments, refined over repeated episodes to maximize a reward function penalizing constraint violations.

### 3.3.3 Advantages and Limitations

ML methods offer compelling benefits:

- **Personalization:** Capture implicit user patterns.
- **Generalization:** Once trained, adapt to new scheduling contexts.
- **Scalability:** After training, produce decisions efficiently.

Yet they also face notable challenges:

- **Data Requirements:** Need large, labeled datasets.
- **Constraint Integration:** Hard constraints are difficult to enforce in learned models alone.
- **Explainability:** Deep models often lack transparency.

## 3.4 Comparison and Justification

When comparing heuristic and machine learning approaches to personal task scheduling, each method reveals strengths that suit different contexts. Machine learning methods excel at discovering implicit patterns and adapting to user behavior over time, especially when trained on large-scale datasets. They handle noise, learn personalized preferences, and enable automation with minimal user input.

Heuristic methods, by contrast, prioritize control, transparency, and efficiency. Schedulers like Squeaky Wheel Optimization allow for direct user feedback, strict constraint handling, and deterministic behavior. These qualities are essential in applications where users need to retain control over how their time is structured.

When evaluating both paradigms, several insights emerge[Sat20]:

- **Execution Efficiency:** Heuristic methods like SWO typically require less time to converge compared to deep RL or GNN models.
- **Interpretability:** Heuristic contacts are transparent and editable by users, unlike black-box ML models.
- **Constraint Fidelity:** Hard precedence, time windows, and spatial constraints are more naturally enforced in heuristics.

Given our requirements for user control, clear constraint handling, and interactive feedback, we conclude that heuristic scheduling, specifically SWO enhanced with preference learning, best aligns with the goals of a practical, user-facing scheduling tool. However, future work can integrate ML components for implicit preference refinement and adaptive guidance.

# Chapter 4

# Theoretical Description of the Algorithms

## 4.1 Overview of Squeaky Wheel Optimization (SWO)

Squeaky Wheel Optimization (SWO) is a heuristic search framework particularly suited for large, complex scheduling problems where complete optimization is computationally impractical. Rather than exhaustively exploring the solution space, SWO iteratively builds schedules by focusing on the most problematic or difficult to schedule activities first.

The SWO cycle consists of three main phases:

- **Construct**: Build a complete solution based on current activity priorities.
- **Analyze**: Identify which parts of the solution are unsatisfactory or conflict-prone.
- **Prioritize**: Reorder the priority queue to address the most problematic elements earlier in the next construction cycle.

This cycle repeats until a termination condition is met, such as a time limit, convergence in utility, or the production of a sufficiently high-quality schedule.

## 4.2 Data Requirements

SWO relies on structured, discrete time representations. In this context, time is modeled as a non-negative integer value starting from 0 [Ref07] and advancing in fixed increments (in this implementation, representing minutes). Scheduling is only allowed in these discrete units, meaning that no activity part can ever take less amount of time than a minute.

Each activity is described by a set of input parameters, which define the bounds and constraints within which the algorithm must operate. These include:

- $duri_{min}$ and $duri_{max}$: minimum and maximum total durations allowed for the activity. These values must be non-negative non-null integers.
- $s_{min}$ and $s_{max}$: minimum and maximum durations for individual parts of an activity. If these values are undefined (null), the activity is treated as non-interruptible.
- $d_{min}$ and $d_{max}$: minimum and maximum spacing between any two parts of an activity. These are also only defined for interruptible activities.
- A set F $\in$ [a$_1$, $b_1$] $\cup$[a$_2$, $b_2$] $\cup$[a$_3$, $b_3$] $\cup$...$\cup$[a$_F$, $b_F$]: disjoint temporal intervals during which the activity can be scheduled. It is important to note that intervals can overlap between activities but not within the temporal domain of a single activity.
- A set Loc: allowed locations where the activity or its parts may take place.

An activity part is formally defined as a tuple, ($t_i$, $dur_i$, $\ell_i$) where $t_i$ represents the start time, $dur_i$ is the duration, and $\ell_i$ is the location. All of the generated parts, alongside the input data must respect all the following defined constraints.

## 4.3 Hard Constraints

The following constraints must be satisfied for a schedule to be considered valid:

**(Cond. 1)** Total duration constraint:

$$\forall A_i, \quad dur_i^{\text{min}} \leq dur_i \leq dur_i^{\text{max}} \quad \text{or} \quad dur_i = 0,$$

*Obs: If $dur_i$ is 0, then that means that the activity has not managed to be scheduled*

**(Cond. 2)** Part duration sum constraint:

$$\forall A_{ij}, \quad \sum_{j=1}^{p_i} dur_{ij} = dur_i$$

**(Cond. 3)** Bounds on each part's duration:

$$\forall A_{ij}, \quad smin_i \leq dur_{ij} \leq smax_i$$

**(Cond. 4)** Minimum distance between parts (interruptible activities):

$$\forall A_{ij}, A_{ik},\ j \neq k \Rightarrow t_{ij} + dur_{ij} + dmin_i \leq t_{ik} \vee t_{ik} + dur_{ik} + dmin_i \leq t_{ij}$$

**(Cond. 5)** Maximum distance between parts (interruptible activities):

$$\forall A_{ij}, A_{ik},\ j \neq k \Rightarrow t_{ij} + dmax_i \leq t_{ik} + dur_{ik} \wedge t_{ik} + dmax_i \leq t_{ij} + dur_{ij}$$

**(Cond. 6)** Travel distance between different activities:

$$\forall l_{ij} \in Loc_i, \forall A_{ij}, A_{mn}, A_{ij} \neq T_{mn}, t_{ij}+dur_{ij}+Dist(l_{ij}, l_{mn}) \leq t_{mn} \vee t_{mn}+dur_{mn}+Dist(l_{mn}, l_{ij}) \leq$$

**(Cond. 7)** Activity parts respecting the temporal intervals:

$$\forall T_{ij},\ \exists k,\ 1 \leq k \leq F_i : a_{ik} \leq t_{ij} \leq b_{ik} - dur_{ij}$$

**(Cond. 8)** Temporal ordering between different activities:

$$\forall A_i, A_j,\ A_i \ll A_j \Leftrightarrow dur_i > 0 \wedge dur_j > 0 \Rightarrow \forall A_{ik}, A_{jl},\ t_{ik} + dur_{ik} \leq t_{jl}$$

**(Cond. 9)** Minimum Activity distance:

$$\forall A_{ik}, A_{jl},\ t_{ik} + dur_{ik} + dmin_{ij} \leq t_{jl} \vee t_{jl} + dur_{jl} + dmin_{ij} \leq t_{ik}$$

**(Cond. 10)** Maximum Activity distance:

$$\forall A_{ik}, A_{jl},\ t_{ik} + dur_{ik} + dmin_{ij} \leq t_{jl} \vee t_{jl} + dur_{jl} + dmin_{ij} \leq t_{ik}$$

The 3 binary constraints are not enforced by the default generation algorithm. Instead, they provide even more freedom for the user to express his preferences and needs properly.

## 4.4 Preferences and Generated Utility

Following that, we have all the types of preferences that the user can select for his activity [RYS10]. These are the main outside source of utility for this algorithm, as they represent soft constraints that are rewarded whenever fulfilled. The first of

them is the activity duration preference, which has the following utility function

$$U_i(dur_i) = \begin{cases} 0, & \text{if } dur_i < dur_i^{\min} \\ u_i^{\text{low}} + \dfrac{(dur_i - dur_i^{\min})}{(dur_i^{\max} - dur_i^{\min})} \cdot 0.8, & \text{if } dur_i^{\min} \leq dur_i \leq dur_i^{\max} \\ 0, & \text{if } dur_i > dur_i^{\max} \end{cases}$$

2 more preferences are allowed, they are related to how early/how late we want our activity to be scheduled inside our system. For that, we implemented 2 more formulas, A piecewise linear function [Rez18] [OBADG07] formula for early, rewarding actvity times that are closer to a certain earlyThreshold:

$$U_{\text{linear}} = \frac{1}{|T|} \sum_{t \in T} \begin{cases} 0.0, & \text{if } t \leq \text{earlyThreshold} \\ 1.0 - \min\left(1.0, \dfrac{t - \text{earlyThreshold}}{\text{last} - \text{earlyThreshold}}\right), & \text{if } t > \text{earlyThreshold} \end{cases}$$

And a stepwise function, which rewards with maximum points anything that comes after the step point:

$$U_{\text{stepwise}} = \frac{1}{|T|} \sum_{t \in T} \begin{cases} 0.0, & \text{if } t < \text{stepPoint} \\ 1.0, & \text{if } t \geq \text{stepPoint} \end{cases}$$

There are 4 more preferences: ordering, minimum and maximum distance, and implication. All of them are binary and thus use a **Degree of satisfaction**.

The degree of satisfaction is a normalized value in the range [0,1] that indicates how well a given preference has been fulfilled. A value of 1.0 means the preference is perfectly satisfied, while 0.0 indicates a total violation. Intermediate values represent partial fulfillment. For each of the 4 preferences.

Mathematically, if a preference P is measured over multiple task instances or parts, the degree of satisfaction can be computed as: [RYS10]

$$\text{DoS}_P = \frac{\text{Number of satisfied cases}}{\text{Total number of relevant cases}}$$

These values are then fed into the utility function to reward the scheduler proportionally. This soft handling allows the SWO algorithm to explore more flexible

solutions while still preferring those that align well with the user's preferences.

Lastly, a distance matrix is provided in which the distance between any 2 locations is calculated and saved. This distance matrix is completed before the beginning of the swo cycle.

## 4.5 Difficulty Metrics in Squeaky Wheel Optimization (SWO)

In the Squeaky Wheel Optimization framework, difficulty metrics are used to estimate the challenge of potentially it is to schedule a task. These metrics guide the initialization and reordering of the priority queue. The main metrics are defined as follows: [RYS10, Ref07]

### 4.5.1 Metric $m_1$: Density of Duration in Available Time

$$m_1(A_i) = \frac{dur_i}{\text{net}(D_i)} \tag{4.1}$$

where:

- $dur_i$ is the total required duration of task $A_i$,
- net$(D_i)$ is the net size of the domain $D_i$, defined as the total number of discrete time units available for scheduling across all allowed intervals.

This metric measures how densely packed the task's duration is within its available time windows. A higher value implies greater scheduling difficulty due to tight temporal constraints.

### 4.5.2 Metric $m_2$: Spread Constraint for Interruptible Tasks

$$m_2(A_i) = \frac{\text{min\_makespan}(A_i)}{\text{width}(D_i)} \tag{4.2}$$

where:

- min_makespan$(A_i) = dur_i + (p_i - 1) \cdot dmin_i$, the minimum span required to schedule all parts of task $A_i$, including the minimum distances between parts,
- width$(D_i)$ is the width of the domain $D_i$, i.e., the difference between the latest end and the earliest start of allowed intervals.

This metric evaluates how well a fragmented (interruptible) task fits within the spread of its domain. A higher ratio indicates a tighter fit, implying higher difficulty.

### 4.5.3 Metric $m_3$: Constraint-Driven Conflict Potential

Though $m_3$ is not expressed as a closed-form formula, it quantifies the risk of conflict due to inter-task constraints, especially with already scheduled or higher-priority tasks. Specifically in our implementation, it is calculated as such: We aim to place as much of the activity inside the available temporal domain until we reach $dur_{max}$. We compute the amount we were able to place and then divide it by $dur_{min}$. the result represents the value of the $m_3$ metric for $A_i$

### 4.5.4 Overall Difficulty

The overall difficulty of a task $A_i$ is defined as:

$$\text{difficulty}(A_i) = \max\left(m_1(A_i),\ m_2(A_i),\ m_3(A_i)\right) \tag{4.3}$$

Tasks with higher difficulty are prioritized earlier in the SWO construction phase, allowing the algorithm to address complex scheduling elements first. If any tasks provide difficulty greater than 1, they are considered unschedulable and ignored in the solution set.

## 4.6 General Structure of the Scheduling Algorithm

### Inputs

The algorithm requires the following input data:

- A set $S$ of activities, each defined by their duration bounds, domain intervals, location options, and potential interruptibility.
- A distance matrix $dm$ encoding the travel times between every pair of locations.
- A list of user-defined binary constraints between activities.
- A set of unary utility sources (e.g., duration preference, early/late scheduling preference).
- A set of binary utility sources corresponding to user-specified preferences.

### Initialization

- For each activity $A_i \in S$, generate a set of unary constraints and unary utility functions.

- For each user-defined constraint between activities, generate a corresponding utility source function.
- Compute difficulty values $m_1, m_2, m_3$ for each activity, and initialize a priority queue $Q$ in descending order of activity difficulty.

## Main Scheduling Loop

Per the general Squeaky Wheel Optimization structure of Construct/Analyze/Prioritize:

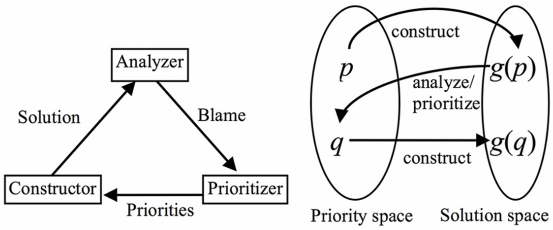


Figure 4.1: The Construct/Analyze/Prioritze cycle

We provide the following loop:

## Construct Phase

- While the priority queue $Q$ is not empty:
  1. Extract the first activity $A_i$ from $Q$.
  2. Depending on the interruptibility of $A_i$:
     - **If non-interruptible**:
       (a) Generate all feasible $(t_i, dur_i, loc_i)$ combinations.
       (b) Apply **forward checking** for each.
           * Remove temporal domains from the remaining activities to ensure respect for all applicable constraints (ordering, min/max activity distance, location compatibility, etc.).
       (c) Select the combination maximizing:

$$\frac{\text{Utility of Scheduled Activities}}{\text{Sum of Difficulty of Remaining Activities}}$$

       (d) Discard combinations that violate constraints or cause any remaining difficulty $> 1$.
     - **If interruptible**:
       (a) Generate part combinations summing to $dur_i^{\min}$.

(b) Forward check each and choose the one with that maximizez the same metric as before.

* Remove temporal domains from the remaining activities to ensure respect for all applicable constraints (ordering, min/max activity distance, location compatibility, etc.).

(c) Attempt to extend toward $dur_i^{\max}$ while maintaining or improving utility ratio once done with providing minimum duration.

3. If no valid schedule is found:
   - Add $A_i$ to the `notScheduled` set.
   - Undo forward checking changes.
4. If successfully scheduled:
   - Move $A_i$ to the `scheduled` set.

## Analyze Phase

- For all remaining activities in $Q$, recompute their difficulty scores $m_1, m_2, m_3$.
- If any activity's difficulty has increased due to the latest scheduling decision, mark it as a *troublemaker*.
- Evaluate the overall utility of the current schedule.

## Prioritize Phase

- Once done with scheduling one activity, reorder the priority queue $Q$ so that:
  - Troublemakers are moved to the front (higher priority) -remaining activities are sorted by updated difficulty scores in the next iteration run.
- Once the queue $Q$ is empty, the total utility of `scheduled` is calculated and compared to the utility of the schedule P, where P represents the current best schedule
- If current schedule utility exceeds that of the best plan $P$, update:

$$P \leftarrow \texttt{scheduled}, \quad \text{UtilityOfP} \leftarrow \texttt{current utility}$$

- Repeat the SWO cycle until no improvement in utility is observed for 10 consecutive iterations.

This algorithm thus manages to generate a locally optimum and correct solution for the provided data.

## Forward Checking

Forward checking, or Propagation [Ref07] represents an action that provides the algorithm the ability to simply ignore any combinations of ($t_i$, $dur_i$, $\ell_i$¿) that could never even happen due to all of the constraints placed on the work domain. What it functionally does is to remove from the temporal intervals of all unscheduled activities the time occupied by the current activity part alongside:

- all of the activity's earlier side of the interval, in case of an ordering schedule
- the minimum distance that is required between any 2 activity parts of 2 activities, in case of a minimum distance schedule

This step represents an important move in regards to the time the algorithm takes to compute. Not only does it have the potential to remove major pieces of temporal intervals, meaning way less combinations of ($t_i$, $dur_i$, $\ell_i$) to check, but it also means that checking all of the constraints will be a task which will pass way more often, meaning less computational work. It also allows us to compute way more accurate difficulty metrics, meaning more activities can be scheduled properly.

## Total Utility

As explained before, utilities come from 3 main sources

- **Respecting the user defined preferences**
- **Respecting the constraints that are imposed upon the scheduling problem (Cond. 1) - (Cond. 10)**
- **Simply placing the activity**

This means that every single one of the constraints placed on the activities will have their respective utility functions which are identical to the ones regarding user preferences. The purpouse of this is to help with the calculation of the final utility, to provide as many sources as possible coming from the hard and soft constraints alongside the activity's placement and duration so that any small change could end up impacting the final result in some way, leading to more dynamic values within the utility computation.

# Chapter 5

# Proposed Architecture

## 5.1 General Structure

Now that we have gone through the general structure of the algorithm we are planning to use, it is time to delve on to the architecture that surrounds it. A system like this should allow users to easily be able to add and edit all of their entities while providing security to their data. Specifically, the application should allow users to:

- Login and Register into the application in order to gain access to their data, so that they can have privacy over all generated schedules and all elements that surround it;
- Perform the CRUD operations on their Activity ideas;
- Perform the CRUD operations on Temporal Intervals for an Activity Idea;
- Perform the CRUD operations and Map certain locations to Activity Ideas;
- Perform the CRUD operations on constraints and preferences related to Activity Ideas, keeping in mind all of their particularities;
- Generate schedules by providing a name and selecting specific Activity Ideas they want included;

For this, the following three-tier system component was implemented:

- A Microsoft Azure-hosted database, ensuring reliable, scalable, cloud-based storage of user data, activity models, and scheduling metadata;
- A Java Spring Boot backend, which encapsulates business logic and scheduling algorithms (including SWO);
- A React [LMT14, ARSP18] frontend, responsible for user interaction and visualization of schedules and any other data;

## 5.2 The Backend

For the backend component of the system, we opted to use the Java programming language in conjunction with the Spring Boot framework. This decision was driven both by Java's robust object-oriented capabilities[CGV20], essential for managing the complex object interactions required by the Squeaky Wheel Optimization (SWO) algorithm, and by Spring Boot's ability to streamline development of web applications.

Spring Boot[Man16] simplifies the creation of standalone applications by offering:

- Auto-configuration, which reduces boilerplate code and enables faster setup;
- Starter dependencies such as *spring-boot-starter-web* (for RESTful API creation) and *spring-boot-starter-data-jpa* (for ORM and data persistence);
- Built-in support for application configuration via YAML or properties files, making it easy to manage environments and secrets;
- Seamless integration with security, scheduling, and validation modules when needed;

The backend is responsible for almost all of the core functionality logic. This includes:

- User Management: Handling registration, authentication (JWT-based), and authorization mechanisms, ensuring that all user data and preferences are securely isolated;
- Conversion between database elements and objects used by the algorithm (for example, converting all of the localDate objects that are stored inside the database into integers, or creating the distance matrix based on the coordinates of the locations);
- CRUD Operations for all core entities aforementioned;
- Schedule Generation Module: A REST endpoint receives a request with selected Activity Ideas and user-defined constraints. It then invokes the internal SWO scheduling engine, which computes an optimal schedule. The output is serialized and stored or returned to the frontend;
- API Exposure: A well-structured RESTful API exposes all necessary endpoints for frontend interaction, using HTTP verbs (GET, POST, PUT, DELETE) and follows RESTful conventions;

The backend communicates with the frontend via JSON payloads over HTTPS and with the Azure-hosted database through secure JDBC connections. Additionally, basic exception handling and data validation are implemented to improve robustness and reliability.

## 5.3 Design Patterns and Technical Principles

### 5.3.1 Design Patterns

The application leverages the Repository Pattern to abstract the data access layer. Each repository interface extends Spring Data JPA's JpaRepository, automatically exposing CRUD operations and enabling custom queries when needed. These repositories are segregated per entity, enforcing the Single Responsibility Principle, and are injected into services to allow decoupling of business logic from persistence logic.

There are also other design patterns implemented accross various subsystems, such as the Strategy Design pattern used for identification of constraint types, unary utility and binary utility sources within the SWO algorithm.

#### Authentication and Authorization

The system uses JWT-based stateless authentication to secure backend endpoints and protect user-specific data. The authentication process begins when a user logs in through the /auth/login endpoint with valid credentials. Upon successful authentication, a JSON Web Token (JWT) is generated and returned to the client.

This token is then included in the Authorization header for all subsequent requests, enabling the system to authenticate the user without storing session state on the server. The application uses a custom JWTAuthenticationFilter, which intercepts incoming requests and verifies the token using the JWTGenerator utility.

### 5.3.2 Service Layer Pattern (Layered Architecture)

The Service Layer encapsulates all business logic for the system. This design:

- Centralizes rule enforcement (for example, validating activity-user relationships before creation);
- Coordinates interactions across multiple repositories (for example, when creating a new activity and verifying related user data);

- Improves testability by allowing business logic to be unit tested independently from controllers or repositories;

### 5.3.3 DTO Pattern (Data Transfer Object)

To avoid tight coupling between frontend-facing request/response structures and internal entity models, DTOs are used to:

- Simplify input validation (for example, expect userId instead of a full User object);
- Minimize over-fetching or over-posting risks in the API;
- Provide clear contracts for each controller endpoint;

### 5.3.4 Dependency Injection

The application makes extensive use of Spring's built-in Dependency Injection (DI) container. Repositories, services, and even scheduling modules are injected into each other using constructor injection, ensuring loose coupling and testability.

### 5.3.5 Modularity and Reusability

The SWO engine is designed as a detachable module, following modular programming principles. It can be reused in other scheduling-based systems with only minimal changes, as long as appropriate adapters are provided to translate input/output data structures.

## 5.4 The Frontend

For the frontend, we have chosen to use the React JavaScript technology due to its modular, component-based architecture and its seamless integration with RESTful APIs. React allows for the rapid development of dynamic, single-page applications (SPAs), which is essential for ensuring a fluid and responsive user experience.

The application is structured around the React Router library, which provides client-side routing functionality. This means that the application can handle multiple logical views (pages) without needing to reload the page entirely. This enhances performance and user interactivity. The routing is configured using the Routes and Route components, which map URL paths (e.g., /login, /generate-schedule) to specific React components that represent each page.

Key routes in the application include:

- / – the Home page
- /login – the Login page, where users can authenticate using their credentials
- /register – the Registration page, for new user account creation
- /activities – the Activity Ideas page, which lists all created activities for the logged-in user
- /schedules - the Page which allows the user to see all of it's former generated schedules
- /generate-schedule – the page used to initiate the schedule generation process by selecting activities
- /generated-schedule – the page that displays the result of the generated schedule

A global authentication context is implemented using React's Context API to manage and distribute the user's login state across components. The AuthContext provider stores the authentication token (authToken) and the username of the logged-in user. This allows components such as the ActivityIdeasPage and GenerateSchedulePage to securely access and interact with protected backend endpoints.

Each major function of the app is represented by a self-contained component or page, promoting modularity and maintainability. For instance:

- The LoginPage handles user authentication, calling the backend and updating global auth context.
- The ActivityIdeasPage fetches and displays a list of activity ideas for the logged-in user, including metadata such as durations, interruptibility, and mapped locations and time intervals.
- The GenerateSchedulePage allows users to select specific activities and provide a schedule name, sending a request to the backend for generation using the SWO algorithm.

Styling is applied using inline styles and CSS classes defined in global.css, with visual design focused on readability and thematic consistency. A soft, modern color palette using purples, golds, and dark blues was used to enhance user engagement and aesthetic appeal.

Overall, the frontend provides a cohesive user interface that complements the backend logic, enabling full interaction with the scheduling system in a secure and intuitive manner. It bridges user input with business logic, enabling task creation, schedule customization, and results visualization through a seamless single-page experience.

## 5.5 The Database (Azure-hosted)

For the persistence layer of the system, we opted for a Microsoft Azure-hosted relational database, leveraging Azure SQL Database for its high availability, automated scalability, integrated security, and ease of management. Azure SQL abstracts away infrastructure concerns, allowing the application to benefit from a managed, cloud-native relational database platform that is both performant and secure.

The schema was carefully designed to support the complex data relationships required by the Squeaky Wheel Optimization (SWO) scheduling engine, while also ensuring referential integrity, modularity, and scalability. The database supports multiple user accounts, dynamic activity modeling, and rich constraint representation, all mapped via normalized relations.

### Core Entities and Relationships

The central component of the schema is the activity_ideas table, which models each individual activity concept a user may want to include in their generated schedule. This table holds all the metadata regarding a single activity that would be neeeded for a scheduler to generate it's result.

Surrounding this core are multiple relational tables that define paramount elements, constraints and preferences, including:

- **Temporal availability:** defined through the temporal_intervals table.
- **Location mapping:** managed via locations and the junction table activity_idea_location, enabling activities to be associated with multiple geographical coordinates.
- **Scheduling preferences:** including duration, distance, ordering, implication, and time of day preferences, each modeled as dedicated tables (for example, activity_schedule_time_preferences, activity_duration_preferences).
- **Constraints:** defined through parallel tables such as ordering_constraints, activities_distance_constraints, and activity_part_distance_constraints.

### Schedule Tracking and Output Storage

Users can generate and persist schedules via the schedules table, which logs metadata such as schedule name and timeframe. Each computed schedule is decomposed into scheduled_activity_parts, which store individual activities with their respective time and location slots, enabling detailed playback and visualization on the frontend. These objects are separated from the activity_ideas table, since they should persist regardless of the state of the objects that were used for generation.

### Technical Considerations

The database is accessed via secure JDBC connections from the Spring Boot backend. This ensures encrypted communication and consistent ORM integration via Spring Data JPA. Azure SQL also provides:

- Automated backups and geo redundancy for disaster recovery;
- Role-based access control and firewall protection;
- Elastic scaling of compute and storage resources;

### Conclusion

Overall, the Azure hosted database acts as the stable, centralized knowledge base of the application, underpinning all business logic and user-specific scheduling functionality. Its integration with Spring Boot allows seamless ORM-based data handling, while the cloud managed nature of Azure SQL ensures robust performance, scalability, and security across all user interactions.

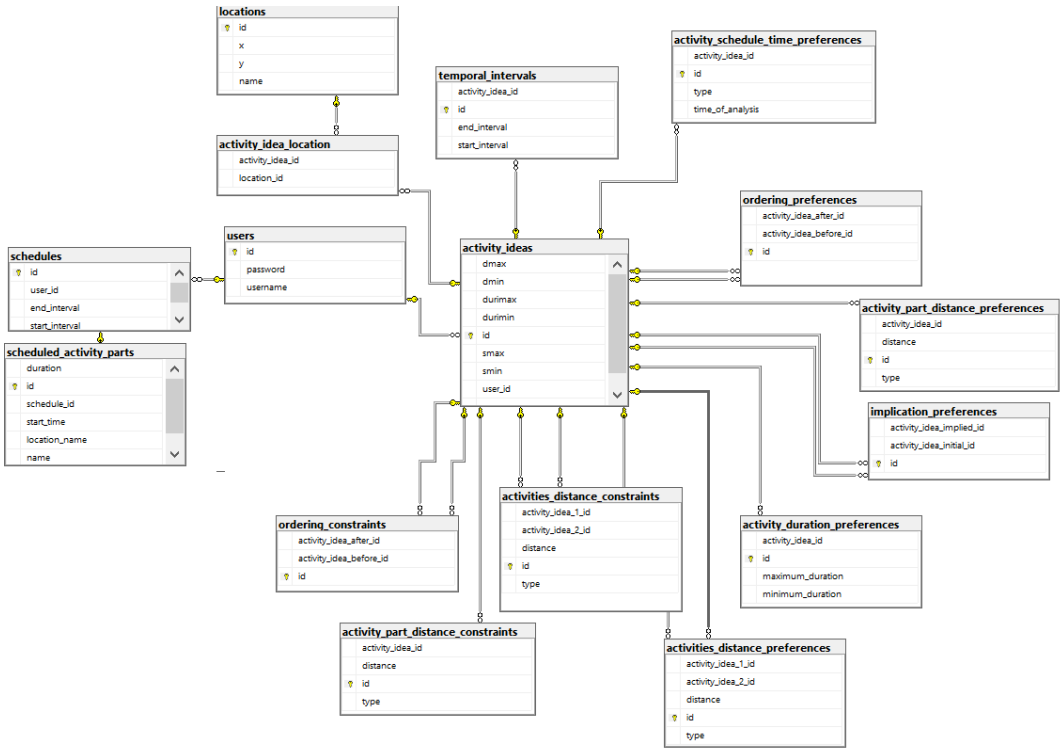


Figure 5.1: The Database structure

## 5.6 Other Diagrams

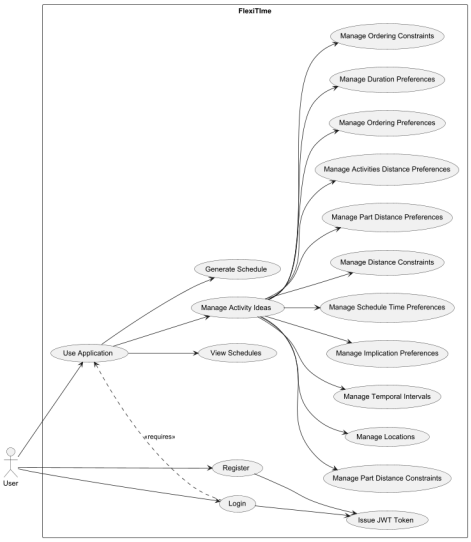


Figure 5.2: Use Case Diagram

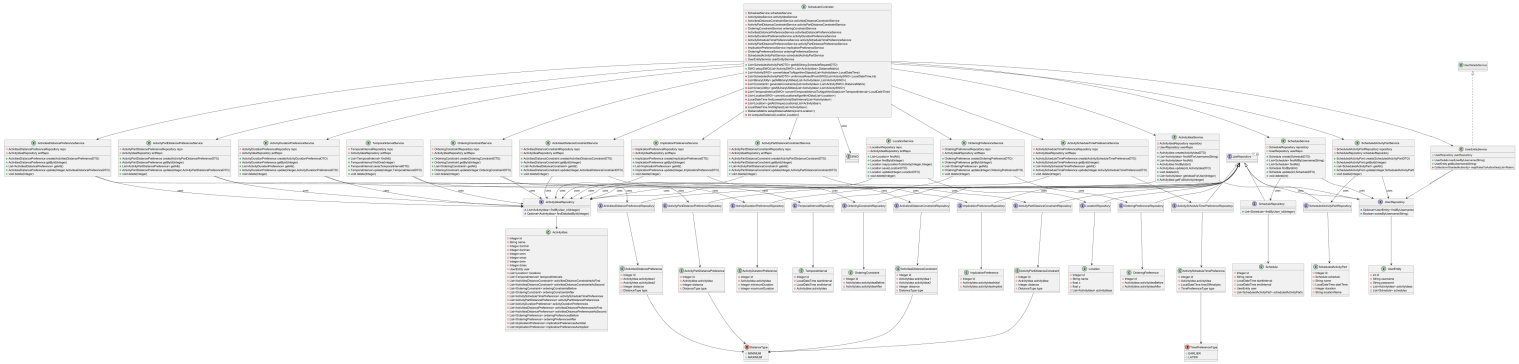

Figure 5.3: The General Structure of the Application (does not contain the controllers for all methods, each one of them is mapped to only its own service)

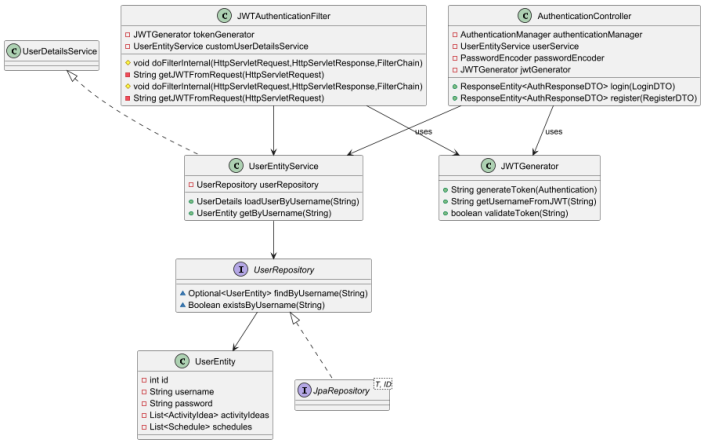


Figure 5.4: Structure of the Security Side of the Backend

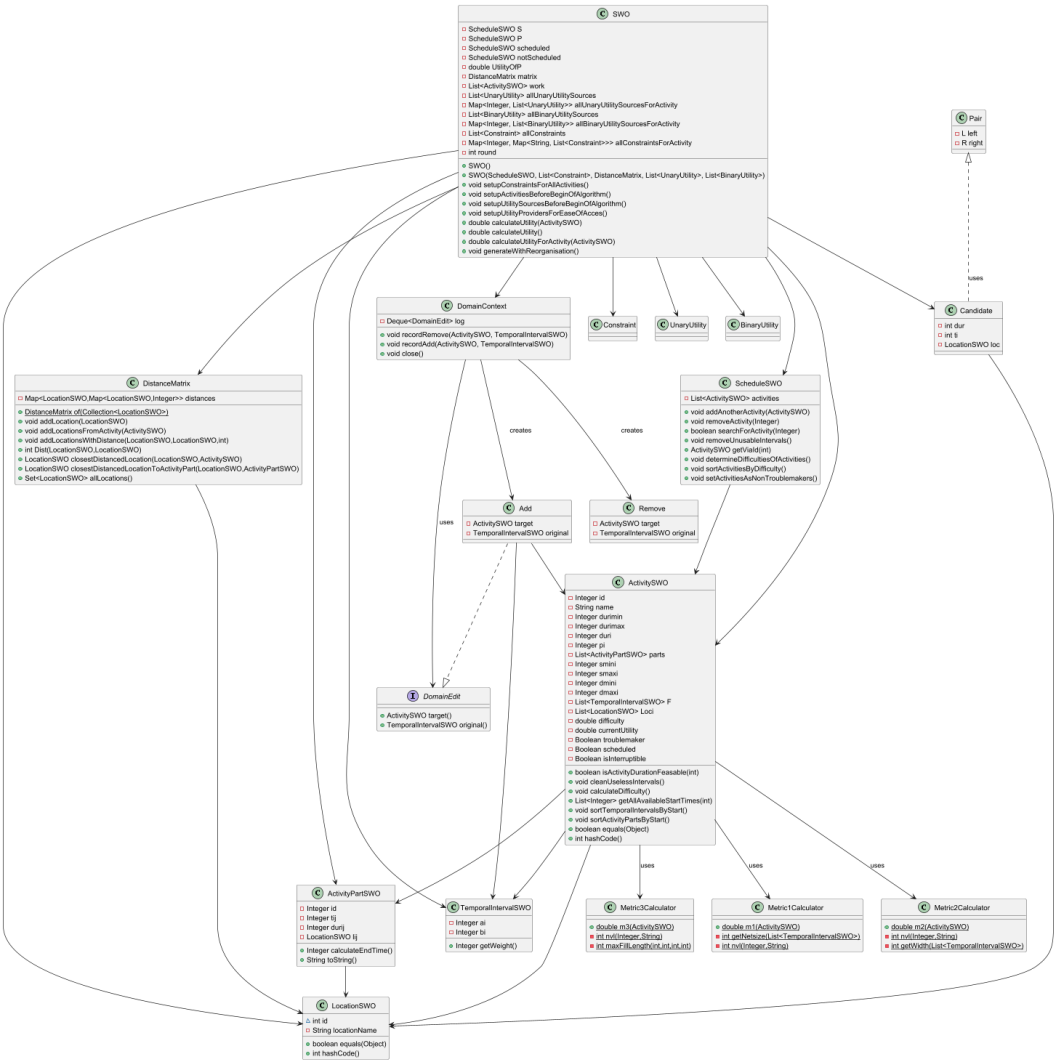


Figure 5.5: Structure of the SWO Algorithm Part of the Code

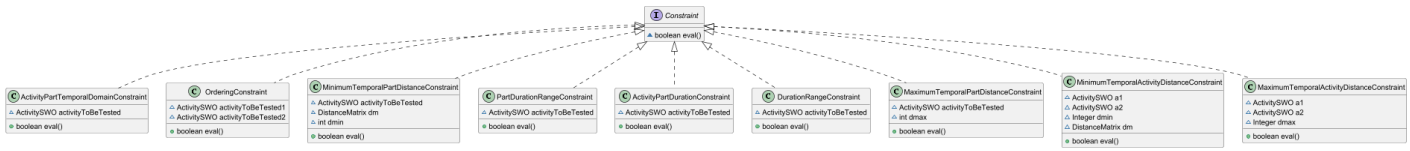


Figure 5.6: The Constraints Structure


Figure 5.7: The Utility Calculators Structure

# Chapter 6

# Experimental Validation

## 6.1 Detecting the best Activity Part placement

In the literature, particularly in the work *A Constraint-Based Approach to Scheduling an Individual's Activities*[RYS10], the authors describe a utility-based scheduling approach where, at each selection of a tuple ($t_i$, $dur_i$, $\ell_i$), the system estimates the utility of all activity candidates left in the priority queue and selects the next activity based solely on the highest estimated utility. While this provides a fast heuristic, it does not account for the increasing difficulty of placing the remaining unscheduled activities as time in the schedule diminishes.

In our implementation of the Squeaky Wheel Optimization (SWO) algorithm, we investigated this shortcoming and explored two alternative utility-driven heuristics that attempt to balance both estimated benefit and placement complexity. Specifically, we compared:

1. **Estimated Utility / Difficulty Ratio (U_est / D)**: This approach selects the activity part that results in the greatest ratio between estimated utility of all remaining activities and the difficulty of scheduling them;

2. **Accumulated Utility + Local Placement Utility / Difficulty Ratio ((U_accum + U_local) / D)**: This approach considers the total utility accumulated so far by the schedule, plus the incremental utility added by placing the current activity part, divided by the difficulty of scheduling the remaining queue.

It is important to mention that no concrete formulas were provided for estimated utility in the found research. Most of them provided suggestions and general guidelines, but never were they explained in a definitive matter. Therefore, our implementation of the estimated utilities took the approach of simply trying to emulate word for word what was relayed in the research paper, meaning that every time we

wanted to create an estimate of utility, we produced a new hypothetical list of activity parts that would allegedly offer the best results, ignoring the constraints placed on the activities. Without any conclusive guidelines on how best to approach this matter, it was the only choice at the time. This information will be important later on.

To initially compare these two approaches, we designed a scheduling scenario involving six diverse activities (A to F) with a mixture of constraints. A brief summary is presented below:

- **Activity A**: Interruptible; $dur_{min} = 3$, $dur_{max} = 4$, $s_{min} = 1$, $s_{max} = 3$, $d_{min} = 2$, $d_{max} = \infty$; time windows: [0, 25], [30, 60]; locations: X, Y.
- **Activity B**: Non-interruptible; $dur_{min} = 2$, $dur_{max} = 4$; time windows: [2, 11], [14, 19]; location: X.
- **Activity C**: Interruptible; $dur_{min} = 3$, $dur_{max} = 6$, $s_{min} = 1$, $s_{max} = 2$, $d_{min} = 0$, $d_{max} = 10$; time window: [3, 19]; location: Y.
- **Activity D**: Non-interruptible; $dur_{min} = 2$, $dur_{max} = 2$; time window: [11, 14]; locations: X, Y.
- **Activity E**: Non-interruptible; $dur_{min} = 1$, $dur_{max} = 1$; time window: [2, 3]; locations: X, Y.
- **Activity F**: Non-interruptible; $dur_{min} = 2$, $dur_{max} = 2$; time window: [17, 19]; locations: X, Y.

Alongside that, we have:

- An ordering constraint $A \ll B$;
- A minimum activity distance constraint between A and B, with a minimum distance between any 2 parts of their activity being 3;
- A piecewise linear utility function which represents the preference that we want Activity A to be placed as early as possible, specifically as early in regards to the threshold 1;

These elements were not entirely selected at random. First, C and B are fighting over extremely similar domains, meaning that the placement of one could dictate the maximum utility the placement of the other could produce. Second, D and F are fighting over even more of B's temporal domain. Activity E is made to work in tandem with the early preference for A, since it explicitely restricts the domain for which that preference could produce max Utility. Finally, the ordering constraint

between A and B means that no activity of A should ever be placed within it's second temporal interval, unless the amount of utility lost would be negligible in the grand scheme of things.

The results are as follows:

Table 6.1: Comparison of Scheduling Heuristics Based on Utility and Difficulty

| **Metric** | **Estimated Utility / Difficulty (U_est / D)** | **Actual Utility / Difficulty (U_accum / D)** |
|---|---|---|
| Total Rounds | 14 | 14 |
| Total Utility | **35.67** | **48.47** |
| Avg. Runtime(ms) | 650 | 570 |

```
Round 14 finished - scheduled order : [5, 6, 2, 3, 1, 4]
Time taken for generateWithReorganisation and output: 687 ms
F : 6
ActivityPart{id=1, startTime=17, duration=2, location=X, endTime=19}
B : 2
ActivityPart{id=1, startTime=15, duration=2, location=X, endTime=17}
C : 3
ActivityPart{id=1, startTime=3, duration=1, location=Y, endTime=4}
ActivityPart{id=2, startTime=7, duration=2, location=Y, endTime=9}
ActivityPart{id=3, startTime=4, duration=1, location=Y, endTime=5}
A : 1
ActivityPart{id=1, startTime=0, duration=2, location=Y, endTime=2}
ActivityPart{id=2, startTime=5, duration=2, location=Y, endTime=7}
D : 4
ActivityPart{id=1, startTime=11, duration=3, location=X, endTime=14}
Total generated utility: 35.666666666666664
```

Figure 6.1: Scheduling output using Estimated Utility / Difficulty Ratio

```
Round 14 finished - scheduled order : [5, 6, 2, 3, 1, 4]
Time taken for generateWithReorganisation and output: 584 ms
F : 6
ActivityPart{id=1, startTime=17, duration=2, location=X, endTime=19}
B : 2
ActivityPart{id=1, startTime=15, duration=2, location=X, endTime=17}
C : 3
ActivityPart{id=1, startTime=3, duration=1, location=Y, endTime=4}
ActivityPart{id=2, startTime=7, duration=2, location=Y, endTime=9}
ActivityPart{id=3, startTime=4, duration=1, location=Y, endTime=5}
A : 1
ActivityPart{id=1, startTime=0, duration=2, location=Y, endTime=2}
ActivityPart{id=2, startTime=5, duration=2, location=Y, endTime=7}
D : 4
ActivityPart{id=1, startTime=11, duration=3, location=X, endTime=14}
E : 5
ActivityPart{id=1, startTime=2, duration=1, location=Y, endTime=3}
Total generated utility: 48.46666666666666
```

Figure 6.2: Scheduling output using Accumulated Utility / Difficulty Ratio

As seen in the figures below, the second strategy not only improved the final utility but also reduced runtime and provided more optimized part placements. This indicates that accounting for the contextual cost of placing future activities (i.e., schedule pressure) results in better global solutions.

Not only that, one more important thing to notice is that the second approach not only manages to schedule the entire activity set, including also E which is absent in the first, but it also manages to schedule more duration for A and C. Therefore, the option to select the local optimum via Accumulated Utility / Difficulty Ratio was selected, since it provided completion and maximum utility of resources.

However, there are some limitations to aknowledge. The implementation of the estimated utility was not efficient, due to a lack of concrete algorithmic solutions, which could have easily led to the average runtime per execution taking the hit it did. Future work could delve further into this concept.

## 6.2 Scaling Behavior of Utility Heuristics on Larger Activity Sets

While the previous experiment evaluated two local optimum selection strategies in a tightly controlled test scenario, we now extend our investigation by analyzing their behavior across a broader and more randomized activity set. This test aims to validate the scalability and robustness of each heuristic when scheduling a large and diverse workload.

### 6.2.1 Test Setup

We generated a dataset consisting of **75 activities** with randomized configurations:

- Each activity was assigned a random duration between **1 and 4**;
- Each activity had temporal availability windows within a range of around **20** time units always located between **0 and 100**;
- Activities were randomly assigned locations among **X, Y, Z**;
- Distance matrix: *dist(X,Y) = 2*, *dist(X,Z) = 1*, *dist(Y,Z) = 3*.

We progressively increased the number of activities from **10 to 75** in increments of 10 (except final step of +5), and measured the performance of the two heuristic strategies described earlier:

- **(U_accum + U_local) / Difficulty**: Our proposed heuristic;
- **Estimated Utility / Difficulty**: Inspired by prior research, using a theoretical best-case utility estimation ignoring constraints.

For each tested number of activities, we recorded the following metrics:

1. Total runtime (milliseconds);
2. Total utility accumulated in the final schedule;
3. Percentage of the total 0–100 temporal interval utilized;
4. Number of activities that could not be scheduled.

### 6.2.2 Results Overview

The following tables reflect the results we received:

Table 6.2: Performance of (U_accum + U_local) / Difficulty Heuristic

| Activities | Time (ms) | Total Utility | % Interval Used | Unplaced |
|---|---|---|---|---|
| 10 | 3395 | 118 | 72% | 0 |
| 20 | 5967 | 394.1 | 67% | 1 |
| 30 | 24658 | 832.5 | 68% | 2 |
| 40 | 18245 | 1214.8 | 73.7% | 6 |
| 50 | 12245 | 1215.4 | 78% | 16 |
| 60 | 12319 | 1359.6 | 80% | 34 |
| 70 | 22302 | 1507 | 75% | 32 |
| 75 | 22212 | 1583 | 74% | 36 |

Table 6.3: Performance of Estimated Utility / Difficulty Heuristic

| Activities | Time (ms) | Total Utility | % Interval Used | Unplaced |
|---|---|---|---|---|
| 10 | 2867 | 97.2 | 57% | 1 |
| 20 | 4442 | 393.6 | 48% | 1 |
| 30 | 11459 | 776.1 | 68% | 3 |
| 40 | 14199 | 1146.6 | 73% | 7 |
| 50 | 37108 | 1356.9 | 79% | 16 |
| 60 | 55368 | 1359.6 | 80% | 34 |
| 70 | 120814 | 1748.2 | 77% | 39 |
| 75 | 151208 | 1506.3 | 74% | 42 |

### 6.2.3 Discussion and Interpretation

The results highlight several key insights:

- **Runtime**: The estimated utility heuristic consistently required significantly more time, especially at scale. At 75 activities, it took over **150 seconds**, compared to just **22 seconds** for the actual utility-based heuristic.
- **Total Utility**: While the estimated utility occasionally produced a slightly higher utility (for example, at 70 activities), it was inconsistent. The actual utility strategy performed more reliably and even produced better results in most cases (for example, at 75 and 60 activities).
- **Interval Utilization**: The actual utility approach generally utilized more of the temporal domain more effectively, particularly at lower to mid activity counts.
- **Activity Placement Success**: In nearly all cases, the estimated utility strategy left more activities unscheduled than the actual utility one. At the 75-activity mark, it failed to place 42 activities compared to 36.

In conclusion, this experiment reinforces our earlier findings: while the estimated utility strategy might occasionally yield a locally optimal arrangement in theory, it struggles with performance, realism, and resource usage in larger, constraint-

rich environments. The accumulated utility approach not only executes faster but more reliably schedules higher-value plans across a wide range of inputs

# Chapter 7

# Testing Methodology and Validation

## 7.1 Overview

Since this application focuses on personalized scheduling logic, user input workflows, and backend correctness the testing strategy prioritized functionality, correctness of algorithm integration, and behavior under realistic application flows rather than benchmarking against large external datasets.

## 7.2 Test Strategy

The testing methodology was divided into three complementary tracks:

- **Application Flow Testing:** The core application workflows, from user registration to activity creation and schedule generation, were manually tested through the frontend interface. This helped identify integration errors between layers and confirmed correct propagation of data and state. Session-based testing proved to an effective tool in this regard.
- **Service Layer Testing:** Unit tests were created for each CRUD operation related to core domain entities such as users, activities, temporal intervals, and constraints. These tests ensured that each service behaves correctly, handles edge cases, and enforces required relationships (for example, an activity must reference a valid user)[PM23]. This test was an important one in order to make sure the business logic of the application worked properly.
- **Heuristic Logic Verification:** Scheduling correctness was validated using custom-built activity scenarios with known constraints and expected outcomes. These scenarios allowed the evaluation of the SWO algorithm's output in terms of feasibility and utility under controlled input configurations. Particular emphasis was placed on testing scenarios where activities had highly constrained

placement options, limited time windows, locations, or duration flexibility, similar to those used in the experimental evaluations. This was done to assess whether the application could effectively adhere to its guiding principle: maximizing the number of successfully scheduled activities.

## 7.3 Test Implementation

Tests were implemented using standard Spring Boot testing infrastructure. Where applicable, service layer methods were tested with mock repositories using Mockito, allowing independent verification of logic without requiring database access.

Key aspects tested included:

- Correct creation, update, and deletion of entities;
- Validation of foreign key relationships and required fields;
- Proper exception handling for invalid operations;
- Consistent transformation of domain entities into SWO-compatible input structures.

## 7.4 Validation Without External Datasets

Due to the personalized nature of the scheduling problem, testing was centered on system behavior rather than statistical generalization. Therefore, no external datasets were created. Instead, hand-crafted input configurations and predefined test cases were used to simulate realistic user interactions and constraint combinations.

## 7.5 Conclusion

The testing approach ensured functional reliability across system components and validated the algorithmic correctness of the SWO-based scheduler. By combining service level unit testing with end to end workflow validation, the system was verified in line with its practical, user-centric goals.

# Chapter 8

# Final Application

## 8.1 Prerequisites to Run the Application

To successfully run the application on a freshly installed operating system, several tools and services must be installed and configured. The application consists of a React.js frontend, a Spring Boot backend using Maven, and a Microsoft Azure-hosted database. The following prerequisites are necessary from the user:

1. **Java Development Kit (JDK):**
   Install Java JDK 14 or later. This is required to compile and run the Spring Boot backend. Verify the installation by running `java -version` and `javac -version` in the command prompt.

2. **Apache Maven:**
   Install Maven to handle project dependencies and build the Spring Boot application. Add Maven to your system's PATH environment variable and verify with `mvn -v`.

3. **Node.js and npm:**
   Install Node.js, which includes npm (Node Package Manager). This is required to run the React frontend. Verify the installation with `node -v` and `npm -v`.

4. **IDE or Code Editor:**
   It is recommended to use an IDE such as Visual Studio Code (for React) and IntelliJ IDEA or Eclipse (for Spring Boot). These tools provide built-in support for project structure, syntax highlighting, debugging, and dependency management.

Once all prerequisites are installed and configured, the application can be started by:

- Running `mvn spring-boot:run` in the backend directory.
- Running `npm install` followed by `npm start` in the frontend directory.

## 8.2 Functionalities allowed by the user

Any user can gain access to all of the data and functionalities of the application by simply logging in with a valid username and password, or registering by creating new credentials. Any other regardless of the register and login is accessed only through a jwt token stored inside the context of the frontend client, only obtainable via the backend. Therefore, any attempts to acces any other routes will lead to empty pages.

By logging, the user gains acces to crud operations for all objects regarding Activity Ideas, their constraints and preferences. Not only that, but the user can generate a schedule by just selecting all of the activities he wants in his schedule, and the algorithm will take all of the constraints and preferences already created and add them to all the scheduling data. The user can then see the result of the schedule on a new page, which he is sent to immediately. He can also revisit all of his old schedules, if he so pleases.

## 8.3 General User Flow

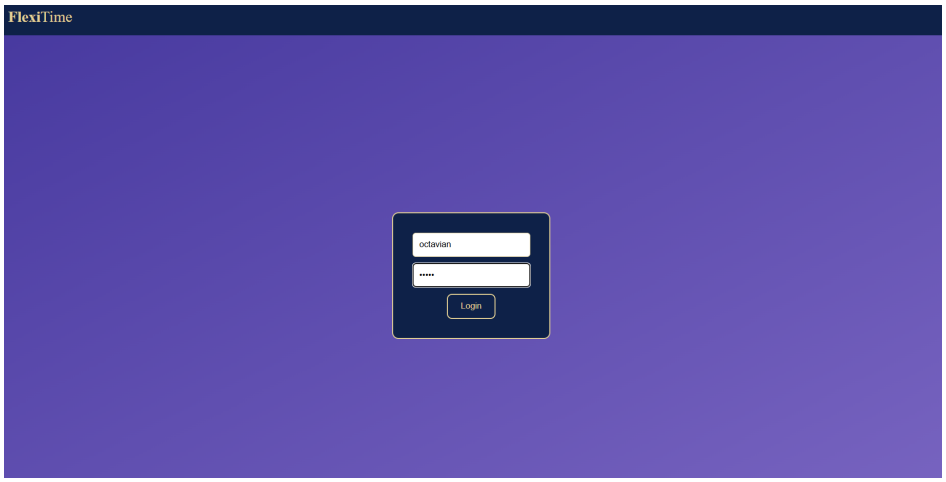


Figure 8.1: Login Screen

After logging in, the user will have a main page featuring 3 options: ”Your Activities”, ”Your Schedules” and ”Generate a Schedule”.


Figure 8.2: The Activity Ideas Page Screen - clicking an activity idea will allow you to edit it's data and the data of the related objects

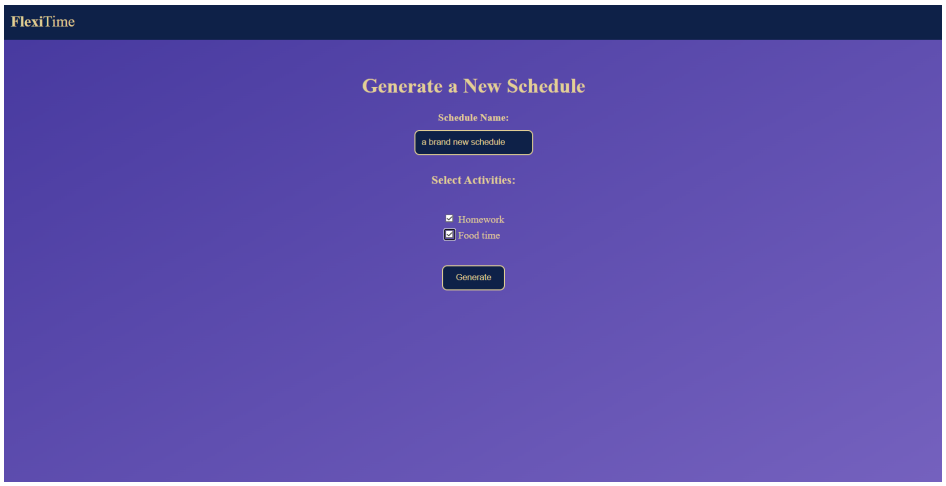


Figure 8.3: The Schedule Generation Page

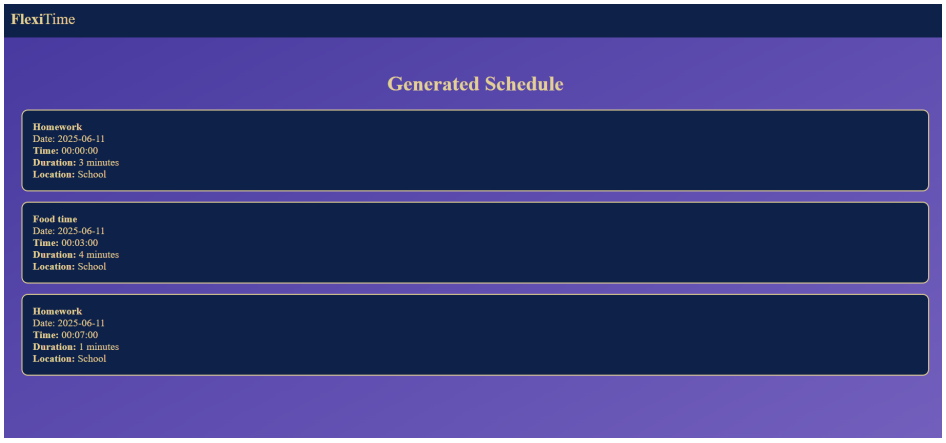


Figure 8.4: The Resulted Schedule

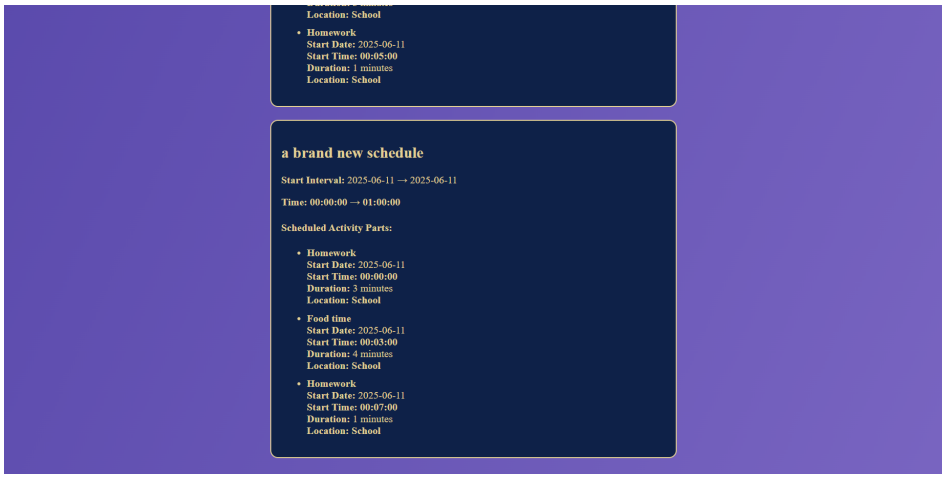


Figure 8.5: The Resulted Schedule

The application's functionalities could increase further on with more developement. Some potential pathways it could take include integration with aforementioned calendar applications such as Google Calendar or Microsoft Outlook, filtering of schedules and manual edit of schedules post generation.

# Chapter 9

# Future Work

While the current developement fulfills the minimal request of creating a smart calendar that uses automatic schedule generation, there are many pathways the project could take if further developed. They mostly go down 2 paths, thos being algorithm wise and application wise:

### 9.0.1 Algorithm Wise

The biggest concept the current implementation of the algorithm is missing is that of estimated utility. No work provides any formulas or concrete and efficient algorithms that could be used to properly estimate the utility an application could possibly receive while keeping track of all of the variables that are at play. Further research needs to be done in order to properly formulate these solutions.

Another thing the algorithm should implement is more proper prioritization of tasks across runs. Currently, the application takes a snapshot of the difficulty of the activities in the priority queue before and after an activity has been scheduled, and places any elements that have increased in difficulty above the current one in the next run. A list of all possible mutations of activity placements is kept, and if a schedule has an identical ordering to a previous one, all the activities are rotated to the right by one position. While this proves to be a working solution, perhaps future work could explore less resource-intensive ways to reschedule activities.

### 9.0.2 Application Wise

While the application serves the purpouse of providing users an interface with which they can play around and add and experiment with their data, their experience would be much enjoyable with some of the following features.

- **3rd Party Service Integration**: Services such as Google Calendar alongside Google Maps would allow users to use their schedules in a more feasible and

interactive way. Providing them the opportunity to export their schedules to google calendar or using Google Maps to define real-world locations would allow them to make better use of all this application and algorithm has to offer;

- **Manual Editing of the schedule**: Further work could allow users to also manually change their schedules once fully developed. This would allow them to make small changes with the result in case something is not to their liking, giving them freedom to augment their data and model it to whatever they desire. Perhaps a system that keeps track of all the changes the user makes to a schedule could be made as well, so that the next time the SWO algorithm runs, extra preferences that simulate the user's behaviour could be made;
- **Mobile Development**: Most Calendar applications also have a mobile counterpart which allows the users to have all the acces to their schedules in their pockets. The implementation of a mobile version of the application could thus be another huge step in having this application become the main hub for all of their schedule-related activities;

# Chapter 10

# Conclusion

This thesis presented the design and development of a full-stack scheduling system aimed at helping users structure their personal activities while respecting a variety of preferences and constraints. By integrating a React-based frontend, a Spring Boot backend, and a Microsoft Azure SQL database, the application offers a robust and scalable architecture suitable for real-world use.

A key focus was on supporting advanced scheduling requirements, such as temporal flexibility, location constraints, and user-defined preferences (for example, ordering, distance, and implications between activities). These were implemented in the backend through structured entities and service layers, allowing the user to create, modify, and manage a rich set of data related to activity ideas and constraints.

Authentication and access control were handled securely using JSON Web Tokens (JWT), ensuring that users could only perform operations on their own data once authenticated. The use of JPA, Lombok, and Maven streamlined backend development, while the React frontend ensured a modern and responsive user experience.

Additionally, a dynamic schedule generation feature allows users to synthesize complete schedules from fragmented activities and constraints. This not only provides functional value but also showcases the extensibility and modularity of the system.

In conclusion, this work demonstrates the feasibility and benefits of building an intelligent scheduling platform that merges heuristic techniques, modular software architecture, and cloud infrastructure. It lays a strong foundation for further development, such as integrating optimization algorithms (Squeaky Wheel Optimization), adding collaboration features, or expanding to mobile platforms. The resulting system stands as a practical tool for structured time management, adaptable to both personal and professional contexts.

# Bibliography


[AR16] Anastasios Alexiadis and Ioannis Refanidis. Alternative plan generation and online preference learning in scheduling individual activities. *International Journal on Artificial Intelligence Tools*, 25(03):1650014, 2016.

[ARSP18] Luciano A. Abriata, João P. G. L. M. Rodrigues, Marcel Salathé, and Luc Patiny. Augmenting research, education, and outreach with client-side web programming. *Trends in Biotechnology*, 36(5):473–476, May 2018. Epub 2017 Dec 15.

[CGV20] Santhosh Chitraju Gopal Varma. The role of java in modern software development: A comparative analysis with emerging programming language. *International Journal of Emerging Research in Engineering Science and Management*, Vol. 1 No. 2 (2020):28–36, 06 2020.

[GKAU24] Esra Gülmez, Halil Ibrahim Koruca, Mehmet Emin Aydin, and Kemal Burak Urganci. Heuristic and swarm intelligence algorithms for work-life balance problem. *Computers Industrial Engineering*, 187:109857, 2024.

[JC99] D. E. Joslin and D. P. Clements. Squeaky wheel optimization. *Journal of Artificial Intelligence Research*, 10:353–373, May 1999.

[KLC+18] Donghyeon Kim, Jinhyuk Lee, Donghee Choi, Jaehoon Choi, and Jaewoo Kang. Learning user preferences and understanding calendar contexts for event scheduling. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, page 337–346, New York, NY, USA, 2018. Association for Computing Machinery.

[LMT14] Kai Lei, Yining Ma, and Zhi Tan. Performance comparison and evaluation of web development technologies in php, python, and node.js. pages 661–668, 12 2014.

[Man16] Ramakrishna Manchana. Building scalable java applications: An in-depth exploration of spring framework and its ecosystem. *International Journal of Science Engineering and Technology*, 4:1–9, 07 2016.

[OBADG07] Amar Oukil, Hatem Ben Amor, Jacques Desrosiers, and Hicham Gueddari. Stabilized column generation for highly degenerate multiple-depot vehicle scheduling problems. *Computers Operations Research*, pages 817–834, 03 2007.

[PM23] Mahsa Panahandeh and James Miller. *A Systematic Review on Microservice Testing*. 2023. Published July 10, 2023.

[PMGM22] Benjamin Platten, Matthew Macfarlane, David Graus, and Sepideh Mesbah. Automated personnel scheduling with reinforcement learning and graph neural networks. In Mesut Kaya, Toine Bogers, David Graus, Sepideh Mesbah, Chris Johnson, and Francisco Gutiérrez, editors, *Proceedings of the 2nd Workshop on Recommender Systems for Human Resources (RecSys-in-HR 2022) co-located with the 16th ACM Conference on Recommender Systems (RecSys 2022), Seattle, USA, 18th-23rd September 2022*, volume 3218 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[RA11] Ioannis Refanidis and Anastasios Alexiadis. Deployment and evaluation of selfplanner, an automated individual task management system. *Computational Intelligence*, 27:41–59, 02 2011.

[Ref07] Ioannis Refanidis. Managing personal tasks with time constraints and preferences. In *Proceedings of the Seventeenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'07, page 272–279. AAAI Press, 2007.

[REK23] Dimitrios Rizopoulos and Domokos Esztergár-Kiss. Heuristic time-dependent personal scheduling problem with electric vehicles. *Transportation*, 50(5):2009–2048, 2023.

[Rez18] Jafar Rezaei. Piecewise linear value functions for multi-criteria decision-making. *Expert Systems with Applications*, 98:43–56, 2018.

[RYS10] Ioannis Refanidis and Neil Yorke-Smith. A constraint-based approach to scheduling an individual's activities. *ACM Trans. Intell. Syst. Technol.*, 1(2), December 2010.

[Sat20] Kaushik Sathupadi. Comparative analysis of heuristic and ai-based task scheduling algorithms in fog computing: Evaluating latency, energy efficiency, and scalability in dynamic, heterogeneous environments. *Quarterly Journal of Emerging Technologies and Innovations*, 5(1):23–40, 2020.

[SH25] Fred Shone and Tim Hillel. Synthesising activity participations and scheduling with deep generative machine learning, 2025.

[Sil04] Edward Silver. An overview of heuristic solution methods. *Journal of The Operational Research Society - J OPER RES SOC*, 55:936–956, 05 2004.

[TEK25] Bladimir Toaza and Domokos Esztergár-Kiss. Assessment of the activity scheduling optimization method using real travel data. *Transportation*, 52(4):1319–1348, 2025.