

SpineWise
Capstone 2 [CIS4951] - Fall 2025
Knight Foundation School of Computing and Information Sciences (KFSCIS)
Florida International University
Instructor: Masoud Sadjadi
Javier Brasil
Oleh Krainyk
Juan Mieses
Jason Pena
John Pena

Executive Summary

SpineWise is a desktop application centered on posture wellness, utilizing computer vision and machine-learning to assist desk-bound users in monitoring their sitting posture and improving it over time. With a basic webcam, SpineWise assesses a user's sitting posture with categorical labels of good, moderate, or bad, logging these trends over time and displaying them in a user-friendly analytics dashboard. This data is then used in determining any potential exercises or stretches necessary for posture health and any assistive apparatuses that can be recommended through the recommendations dashboard. This project has been ongoing since Capstone I of Summer 2025, where the main backend of the code was realized: a rules-based approach using OpenCV and MediaPipe pose landmarks to detect posture deviations. Capstone II of Fall 2025 built upon this foundation by expanding on the GUI shell created in the initial semester and overhauling it, adding LightGBM machine learning onto the posture detection with our own personalized dataset, and expanding upon the voice control system with adjustable voice commands and multilingual support. The machine learning engine improved moderate posture detection and greatly increased the accuracy of the system up-close, while introducing sensitivity when the camera was distant. SpineWise does not exist as a replacement for physical therapy, treatment, or even diagnoses of posture issues; it is but one low-cost solution that aids in the wellness and productivity of its sedentary target audience by guiding users towards healthier habits and allowing for the self-monitoring of their health in a tailored, yet privacy-conscious approach, with more accurate posture data granted as of recent by way of our machine learning model.

Table Of Contents

Executive Summary.....	1
Problem & Requirements.....	3
System Overview & Architecture.....	4
Algorithms & Data Structures.....	5
<i>Pose & Landmark Extraction</i>	6
<i>Posture Features & Calibration</i>	7
<i>Scoring System</i>	7
Implementation.....	9
<i>Codebase Organization</i>	9
<i>Threading & Optimization</i>	9
<i>Rationale For Design Choices</i>	10
Testing & Evaluation.....	10
<i>Part 1: Rule Based</i>	10
<i>Part 2: Machine Learning</i>	11
Security, Privacy, Accessibility & Compliance.....	12
Deployment & Operations.....	14
<i>Installation Guide</i>	14
Limitations & Future Work.....	15
References.....	16
Appendices.....	17

Table Of Figures

Figure 1: <i>System Architecture Diagram</i>	4
Figure 2: <i>Class Diagram</i>	6

Problem & Requirements

In the working world, whether in an office setting or remote, health degrades as one remains sedentary for long periods of time. One such area of health that suffers from being stationary like this is spinal health, where people start holding invisibly uncomfortable positions and get in the habit of sticking to these “shrimp-like” positions of slouching that cause back and neck pain. Whilst this can be remedied by self-regulated breaks consisting of stretching, movement, and posture re-correction, many individuals—whether student or worker—do not take these measures for they simply forget. Ergo, SpineWise exists as one solution to this varied problem; by tracking posture data in real-time against our own dataset of varying postures while one sits, the program can detect bad posture, analyze problem areas within an end user’s sitting habits, and recommend exercises or products to further improve their posture health. Furthermore, the application is relatively discreet during operations, where it must operate in the background, and has modules for hand-free usage. Our stakeholders are members of the general public who spend most of their time in front of a computer for work, studying, or other careers of stationary work: software developers, students, accountants, to list a few.

Requirements

Functional Requirements:

- Real-time posture detection.
- Analytics tab for easy logging.
- Alert system for users to correct posture at a moment’s notice.
- Recommendations system for live coaching and product recommendation.
- Modular settings that can be tailored for user comfort and accessibility.

Non-Functional Requirements:

- Must run in the background with minor interference and resource consumption.
- Must run on most consumer desktop/laptop electronics.
- Must work with most standard webcams.
- Offline functionality.
- Privacy and security assurance.
- Optimization to avoid crashes, lag, and general performance issues.

Constraints:

- Low/non-existent budget.
- Limited time for project delivery and quality assurance.
- All members are full-time students and active employees.

Success Criteria:

- Capable of delivering accurate results during assessment of posture.
- Alert delivered *immediately* after several seconds of poor posture maintenance.
- Ensuring low memory/CPU usage when program is active.
- Can run in the background for hours without issue.

System Overview & Architecture

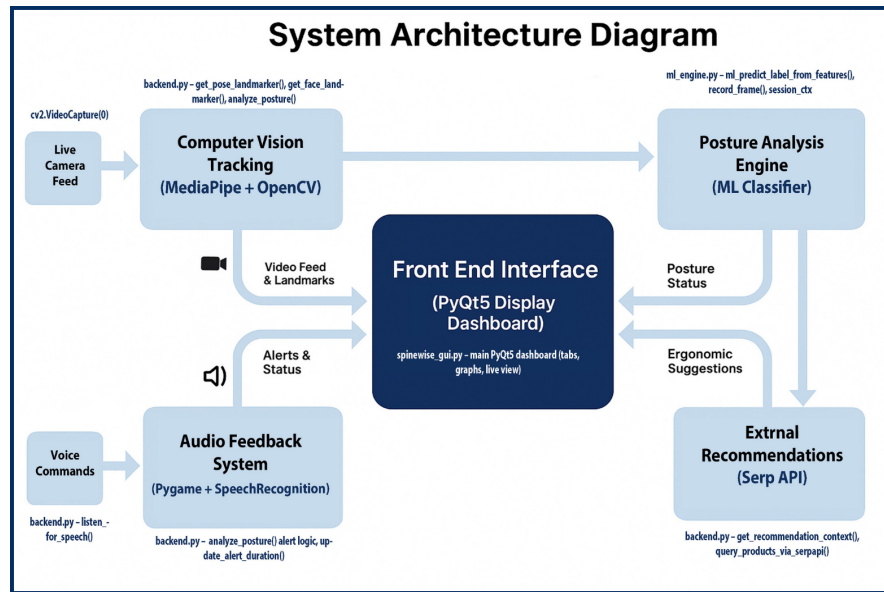


Figure 1: System Architecture Diagram. Main components are displayed involving real-time posture monitoring: PyQt5 front-end interface, computer vision (OpenCV + MediaPipe), posture analysis engines (rule-based + machine-learning), audio alert system (PyGame), voice recognition (SpeechRecognition), and recommendation system (SerpAPI). Omitted for space, Algorithms & Data Structures describes .csv/.json logging.

SpineWise is a Python-based desktop application, using Layered Architecture (see Figure 1), with clear layer separation between:

- GUI layer for user interaction, settings configuration, and data logging (PyQT5) [1].
Comprised of five tabs:
 - Dashboard - Live camera view and posture status display, alongside buttons for enabling/disabling camera, and calibration controls.
 - Analytics - Short-term posture trend log across recent sessions, aggregated weekly/monthly summaries as well as a streak tracker, and session duration statistics, among others.
 - Recommendations - Displays a table of recommended products, their names, prices, etcetera, and a live coach panel of exercises based on the most dominant posture issues.
 - Settings - Allows for notification adjustments on volume, frequency, duration; allows for configuration on voice settings and language; allows for displaying landmarks for debugging, and changing posture detection method between ruleset and machine learning.
 - About - Credits that introduces each member both past and present.
- Backend layer that handles camera capture (OpenCV), pose/facial landmark extraction (MediaPipe), posture scoring, stability logic, logging, and alerts (PyGame) [2, 3].
 - Frame capture by OpenCV, MediaPipe landmark extraction from frames.

- Computes posture-related features and compares them against a user-tailored *good posture* baseline that was *calibrated* for.
- Classifies posture findings against *rule-based* or *machine-learning* engine.
- Logs posture events and triggers audio alerts when bad posture persists beyond threshold.
- Machine learning engine layer that analyzes short windows of posture features to categorize posture and infer specific problematic habits.
 - Aggregates sequences of per-frame features into window-level statistics; $O(w*f)$ (window length by feature amount).
 - Uses LightGBM model and calibration layer for posture classification [4, 5].
- Additional modules for data logging, dataset generation, voice commands, and UI theming.
 - Voice commands using speech recognition library with multiple languages and modifiable activation phrases [6].
 - Recommendation module utilizing SerpAPI for shopping queries from posture pattern score data [7].
 - Relevant data is stored in .csv and .json files locally on the user's computer.

SpineWise runs locally in its entirety and no data is streamed or uploaded aside from SerpAPI queries for product recommendations based on generic search terms related to user posture problems. This layered architecture format ensures separation of tasks for optimized workflow that can be added onto and expanded in future iterations.

Algorithms & Data Structures

At the core of SpineWise lies a foundation of classical computer vision and a scoring system based on our custom ruleset for monitoring posture, which then led to machine-learned classification. This section elaborates upon the various algorithms that comprise the code and the choices made behind them. Next page illustrates Figure 2, a Class Diagram showcasing the code and its internal relationships.

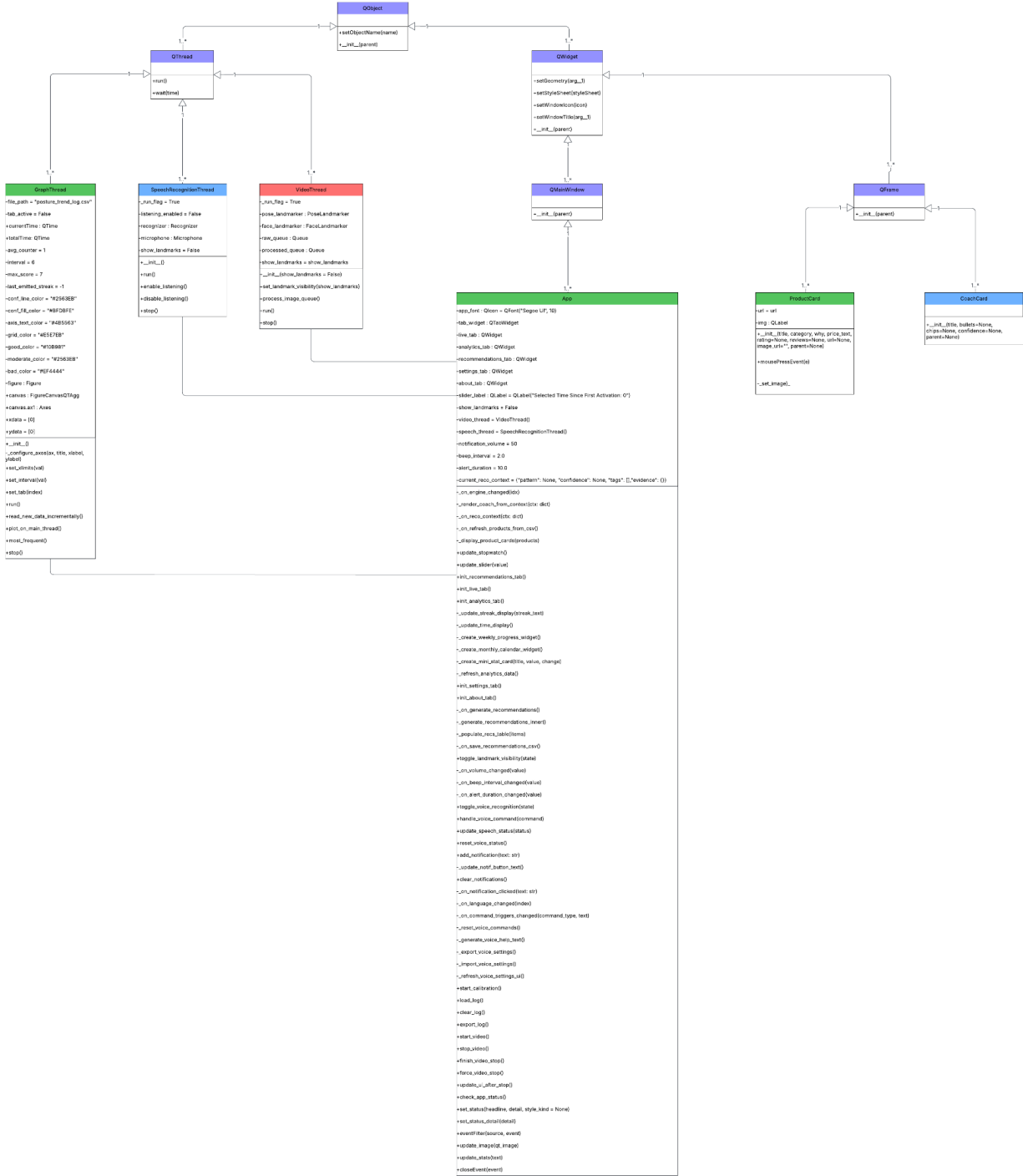


Figure 2: Class Diagram of SpineWise code elements.

Pose & Landmark Extraction

The foundation of the SpineWise system lies on top of MediaPipe and OpenCV. MediaPipe's pose and face landmarks are used to extract 2D image coordinates and 3D coordinates from these pose nodes from each webcam frame. The sequence of processing these frames runs like this:

- Webcam & OpenCV Initialization: OpenCV captures feed frames and converts it to a computable format for MediaPipe.
- MediaPipe Pose Features: The MediaPipe model returns the base bodily landmarks of shoulders, nose, and hips (optionally) among others, and the facial landmarks of cheeks, eyes, facial contours, etc.
- Normalization: These landmarks are normalized to the image size and scaled for sensitivity reduction in regards to resolution influence.

This outputs, per-frame, a dictionary of numeric features derived from the aforementioned landmarks rather than the raw landmark arrays, reducing memory usage and simplifies models later on.

Posture Features & Calibration

The frame outputs are logged into a CSV file for rules-based scoring and machine-learning. The features within those frame outputs are:

- “head_tilt”:
- “clavicle_drop_pct”:
- “face_lean”:
- “shoulder_ear_pct”:
- “torso_lean_pct”:
- “looking_down_pct”:

These features are aggregated during *calibration*, where a rolling window of frames are captured whilst the user holds their best posture possible. The running means and medians of these landmarks distances and angles are stored as a threshold for relative comparisons. This per-session tailoring of posture baseline ensures that sessions are contextually relevant to where the user is positioned, and ensures the user isn’t penalized for natural asymmetries of their head tilt or shoulder. The calibration data structure is a small, immutable profile object used by both engines (rule-set and machine-learning) as a baseline reference for proper posture.

Scoring System

SpineWise divides its running engine in two modes, configurable in the settings tab: *rule-based*, and *machine-learning*. In *rule-based* mode, SpineWise uses a deterministic scoring function that maps the per-frame feature vector into a discrete posture label.

- Piecewise threshold functions are applied to each continuous feature for a per-feature score on a small integer scale (0-7 confidence, 0 being unproblematic, 7 being the maximum amount of problematic)
 - Per-frame processing is $O(n)$ for the number of landmarks, constant.
- Certain features are weighted more heavily, such as clavicle drop and torso lean, for greater penalizations for slouching over minor head movements.

- Sums and normalizes these scores for a composite posture score as mentioned in the first bullet. Thereafter, the scores land in one of three categories:
 - *Good*
 - *Moderate*
 - *Bad*

Faster performance and simplicity made *rule-based* mode perfect for data collection for our first few sprints, and its simplicity ensured consistency across camera distances. Optimization for tracking stability required a layer that utilized a sliding ring buffer window of recent posture predictions and their confidence scores, and a simple state machine with a minimum frame requirement before changing posture states and early confirmation of transition from good to bad when composite scores rank beyond the threshold. The ring buffer's format is a deque of tuples (timestamp, label, confidence), where updates recomputes the dominant state over the recent window and commits to that state if it satisfies transition rules. This ensures smooth transitions between posture ranks instead of flickers that may ruin data collection and analysis, and the operation is $O(1)$ per frame.

Machine-learning mode is a Capstone 2 implementation that uses LightGBM classification trained on short windows of posture and a meticulously gathered and varied dataset. The pipeline sequence is as follows:

- Frame buffering where frame-level feature dictionaries are appended to a per-session buffer. A windowing mechanism slices overlapping windows of fixed duration with timestamps to maintain consistent temporal spacing.
- Each window is aggregated into summary statistics for all frame level features, and their means, min/maxes, standard deviation, range, and other trends. This is far more dynamic than rule-based.
- LightGBM multi-class classifier runs the aggregated vector through and outputs raw scores for posture classes, which are then turned into calibrated probabilities, which *then* undergoes a decision function in determining the final label that greatly penalizes missing true “bad” postures while allowing uncertainties between “good” and “moderate”.

At the cost of performance (a mild latency issue with the camera, which results in a 2 second delay), the *machine-learning* engine is a dynamic, more accurate improvement for a majority of users, ones that are up-close to their laptops while displaying all relevant nodes within frame.

Z-Score norms are utilized to estimate severity of patterns, which allows for the recommendations tab to determine exercises and products for the user to receive as suggestions. SHAP (Shapley Additive Explanations) is used as well, in order to map numeric feature deviations to human-readable posture problems [8]. These problems are grouped into semantic patterns based on relation, such as head-position features put into a *forward_head* group, and SHAP values are used to determine the contribution of each feature to the current assessment. These are aggregated for each feature group and combines them with z-scores to compute “need scores” (determining which feature is in need of recommendation/live coach assistance). This

generates a dictionary of pattern scores for utilization by the recommendations module, where SerpAPI shopping queries are fetched by relevance to posture patterns.

For logging, *posture_trend_log.csv* and *stats.json* stores relevant data: the former being a chronological log with timestamps, posture labels, and selected feature values and scores, and the latter is a dictionary made to determine streak and posture summaries for the analytics tab. The .csv is presented as Pandas DataFrames.

Implementation

SpineWise's implementation aims to be a modular Python codebase as per the layered architecture mentioned on page 3. The code's organization was meticulously planned and twice overhauled to ensure split responsibilities between modules for optimization and programming legibility.

Codebase Organization

The main structure is set-up where:

- *spinewise_gui.py* is the main GUI script that initializes the program itself, the PyQt5 application window and its tabs, and the modules that encompass the tabs (analytics, recommendations, about).
 - *spinewise_theme.py* maintains consistent theming for the GUI, put in its own file for decluttering.
- *backend.py* provides core runtime logic: webcam capture, landmark extraction, feature assessment and classification, calibration, stability and optimization, logging, audio alerts, and recommendation calls for SerpAPI.
- *ml_engine.py* contains the LightGBM model for machine-learning assessment of posture, with aggregation and scoring.
- *voice_config.py* stores settings and additional parameters for speech recognition, with handling custom commands and additional languages.

Threading & Optimization

The main feature of real-time video processing and the optional feature of voice recognition are notoriously CPU-heavy tasks, and were the source of many crashes as the program underwent development. Responsiveness had been maintained with:

- PyQt5's main thread utilizing event handling parameters and interface drawing, keeping the GUI call light.
- Backend optimization by ensuring the webcam capture and the posture analysis utilized thread-safe mechanisms whenever they periodically pushed updates on their perpetual loops.
- Voice recognition, GraphThread, and SerpAPI calls underwent various rewrites to ensure crash prevention.

This prevents UI freezes/hangs, camera freezes (which could lead into crashes), and potential hangs with GraphThread or SerpAPI. Generally, application design is extensively linear and is effectively constant-time per frame, and the program functions at a locked 30FPS.

Rationale For Design Choices

On the basis of ethics, SpineWise opted out of online services for this stage of the project. While this is primarily related to scope as time and resources were too low to consider implementing this in full, being a local desktop application avoids the ethical pitfalls in transmitting webcam data or *any* user data over a network and dispensing queries. It also simplifies the system in how data is stored in .csv and .json files, making set-up easier for most users.

LightGBM versus more complex models is due to efficiency and appropriateness; for the data handled, there is little need to be convoluted in our approach, for this potentially sacrifices system performance more if the models were to be too complex. LightGBM also operates well with SHAP with its tree structures, and is more appropriate for window-level features. Keeping the rule-based engine is also for the sake of performance and allowing user choice in tailoring the system to their needs; as of now, the machine-learning engine still has some latency issues prevalent in some machines, which some users may not desire, even if results are more accurate.

Testing & Evaluation

Testing was manual, scenario-based, with a focus on end-to-end behavior.

Part 1: Rule Based

During one of our testing phases, the goal was not to classify posture into perfect categories but instead to evaluate how reliably the system can detect posture changes under real-world conditions using the rule based model. For this specific testing we defined success as when the user intentionally changes posture from good to bad or bad to good, SpineWise should correctly leave or return to the appropriate posture state. In this case “moderately bad” is counted as correct during intentional slouching because it represents posture degradation. In other words this evaluation focuses on transition reliability rather than strict classification.

Testing Methodology:

We recorded several short, controlled sessions where the user alternated between very obvious good and bad posture:

- *Good* posture (upright, neutral head position, shoulders visible)
- *Bad* posture (intentional slouching, rounded shoulders, head drop)

Each posture phase lasted about 15-30 seconds. We produced an annotated spreadsheet with:

- Condition
- Timestamp (coming from posture_trend_log.csv)
- SpineWise classification (coming from posture_trend_log.csv)
- Ground truth posture (users posture change)
- Correct/incorrect

- Total rows
- Correct rows
- Accuracy

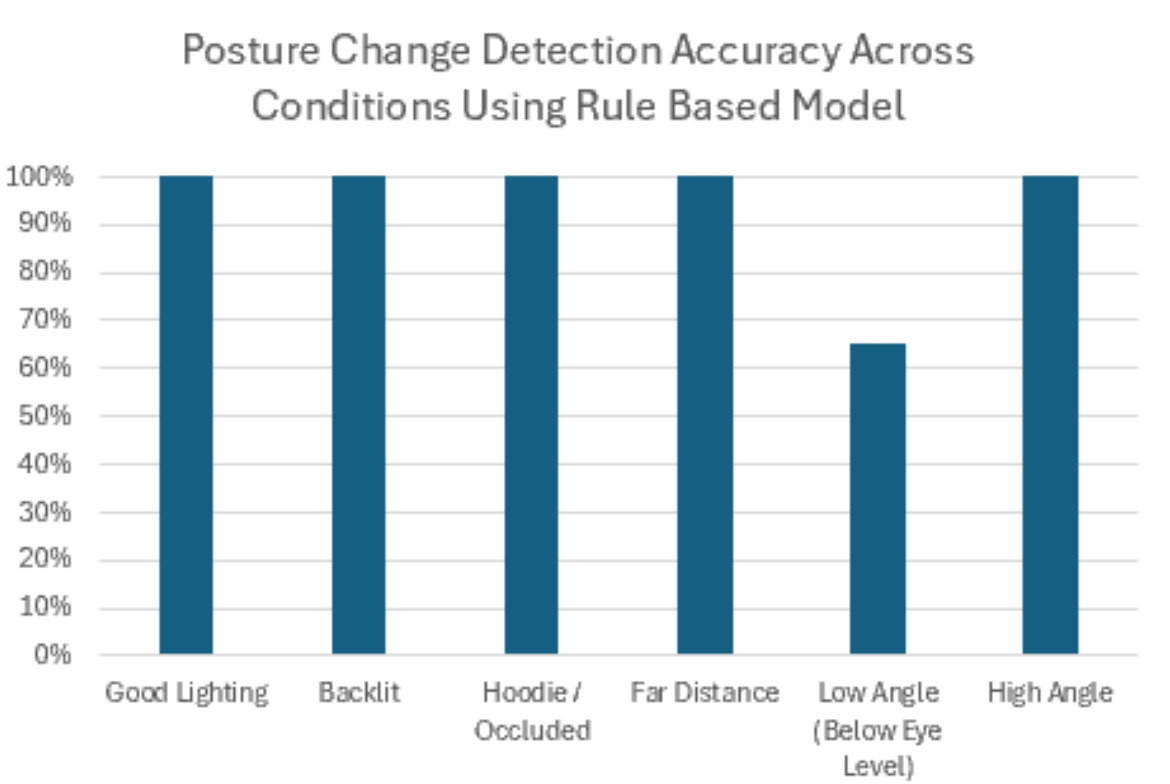
The Environmental Conditions Tested:

We evaluated Spinewise under six user scenarios:

1. Good lighting(baseline)
2. Backlit(strong light behind user)
3. Hoodie overhead
4. Camera placed further away(around 3 feet)
5. Camera placed significantly below eye level(on lap)
6. Camera placed above eye level

Results:

Transition-accuracy remained high across most conditions. Only conditions involving camera angle distortion like camera below eye level making it hard to distinguish the angle of the shoulder and head produced a drop in accuracy.



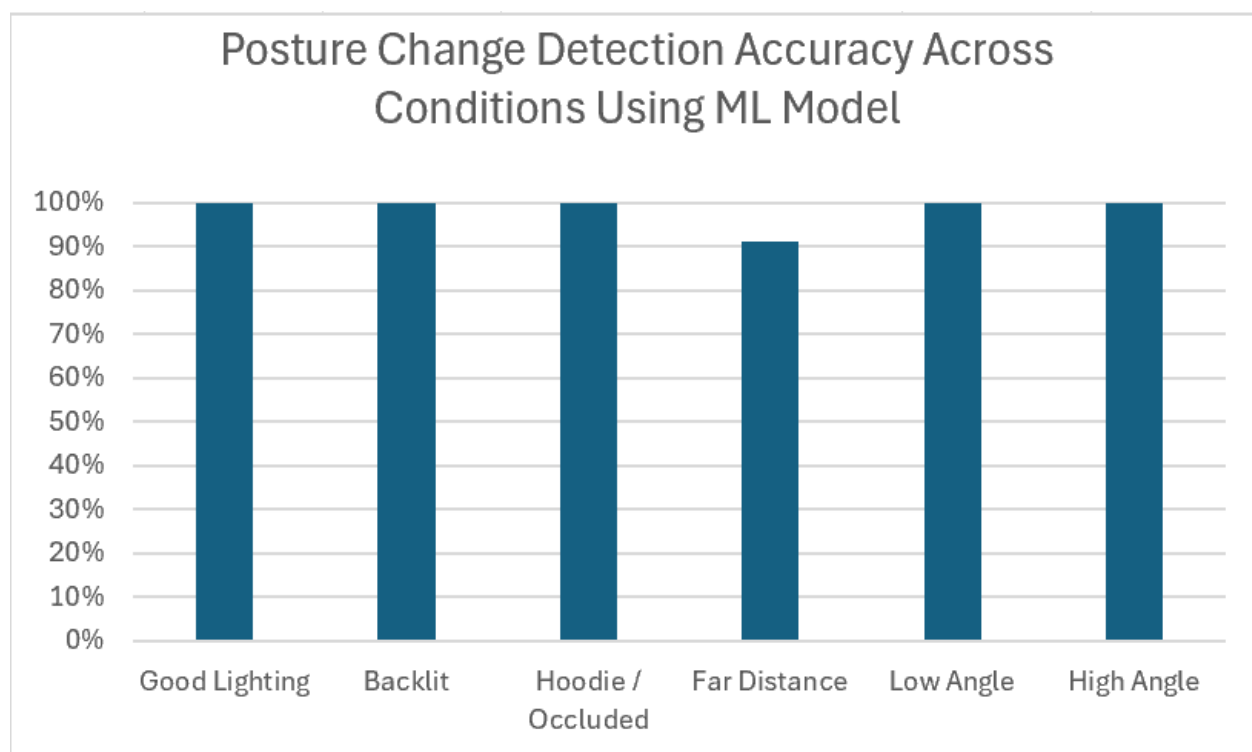
Part 2: Machine Learning

During our second testing we followed the same evaluation method as the rule based model, however, using our new machine learning model. We again relied on the use of posture_trend_log.csv for accurate timestamped posture classification. Our goal was to verify

whether the ML model could accurately detect posture changes in harsh real world conditions when the user is intentionally changing between good and bad posture (“moderately bad” being counted as bad).

Results (ML model):

Compared to the rule based system, the ML model was able to more accurately predict the correct posture when purposefully pivoting. This becomes obvious when specifically looking at the low angle scenario where the rule based approach struggled due to distorted head and shoulder ratios. This is most likely due to the ML model learning multi dimensional spatial trends. Although it excelled in the low angle portion, the distance portion seemed to have made it mildly uncomfortable. When the camera was positioned farther (about 3 feet away) the model began to interpret neutral posture as deteriorating posture. This could suggest that the decreased clarity of facial and shoulder landmarks might have degraded its ability to identify subtle changes. In summary, the ML model reduced angle related misclassification but introduced new misclassification to distance and scale.



Security, Privacy, Accessibility & Compliance

Security

- Local Video/Audio Processing

- All handling of video and audio is local-only; no history is made of audio transcripts, no data of video and audio are sent to external servers for processing. Landmark data is only derived numerically.
- Imageless Logging
 - No visual imagery of camera are saved.
- Limited Network Calls
 - Outbound requests are limited to downloading MediaPipe model assets and calling for SerpAPI shopping queries with generic terms only and no additional, potentially sensitive information.

Privacy

- Offline Functionality
 - No account system, no cloud storage, and no uploading of files prioritizes local operation of software and ensures all data remains on user machines.
- User Camera Control
 - Users can toggle camera and disable it at any time.

Accessibility

- Voice Command
 - Speech recognition allows for hands-free operation of start, stop, and calibration functions for users with mobility limitations.
- Multilingual Voice Support
 - Settings includes multiple recognition languages for non-English speakers.
- Visual Design
 - High-contrast text and background colors ensure visibility and understanding for all seeing individuals. Buttons have visual feedback for hovering and clicking, and buttons are large and well-padded for users who may struggle with mouse control.

Compliance

SpineWise is not medically approved and thus does not collect or require the use of regulated health information such as medical records, and does not integrate with clinical systems. HIPAA and medical compliance is not required or necessary for this project, but the project still complies with general privacy principles: data minimization, local processing and handling, and user control over camera and logging. If deployed institutionally or otherwise, it would have an appropriate privacy notice for its camera, even if no actual image and video information is recorded.

Deployment & Operations

Platforms

Minimum Requirements:

- OS: Windows 10/11, macOS Sonoma 14.3+, Linux Ubuntu (recommended)
- RAM: 8GB
- Storage: 2GB
- CPU: Multi-core processor
- Peripheral Devices: Webcam (Integrated or external), microphone (optional)

Runtime Dependencies

- Python 3.11+.
- *Core libraries:*
 - OpenCV (cv2)
 - MediaPipe
 - PyQt5
 - Numpy, Pandas
 - LightGBM, Optuna, scikit-learn, joblib, SHAP
 - SpeechRecognition, PyAudio
 - Pygame
 - SerpAPI

Installation Guide

1. Clone the repo to local machine.
`git clone https://github.com/SlyJavii/SpineWise.git`
2. Navigate to repo directory and create a virtual environment.
`cd SpineWise`
`py -m venv .venv`
3. Activate virtual environment.
 - a. MacOS and Linux only:
`source .venv/bin/activate`
 - b. Windows:
`.venv\Scripts\activate.bat`
4. Install SpineWise requirements
`py -m pip install -r requirements.txt`
5. Launch spineWise_gui.py
`py spineWise_gui.py`

Limitations & Future Work

Performance issues with memory and CPU usage leaves uncertainty as to whether or not this application is at an optimized enough state for being deployed, especially with how the main purpose of the program—machine-learning posture detection—causes latency lag on an active camera, thus defeating the purpose of our *real-time* posture logging. Concerns stem from the fact that productivity applications may fight with SpineWise, a constantly-running background application, for resources, and this could result in potential crashes or freezes. During testing, these didn't occur, but with different set-ups it is possible. The pipeline would require further optimization with models, scheduling, and threading to guarantee low crash/bad performance risks. Future work would entail solving these issues and separating the codebase further into separate files; in retrospective, `spinewise_gui.py` became too massive to handle feasibly and all these helper functions in one file is illegible and leaves the code unnavigable, harming our modularity by creating clutter. Further optimization is needed for this code to be proper, which is attainable, but not within the time constraints the team underwent.

Accessibility leaves much to be desired. Testing with screen readers wasn't possible due to time constraints, and would need additional effort to verify. Although colors and text pairings improve accessibility, themings for dark/AMOLED or high-contrast was not feasible in our scope and leaves out portions of target audience who may need those. Additionally, whilst there is the multilingual facet of the voice configuration module, there are no built-in commands for each language as a preset, which requires manual input from end user, and the application does not have alternate languages for text, making the multilingual module inconsistent against the rest of the application.

Whilst keeping the application a local affair ensures privacy and security, users cannot track history between devices. The analytics tab is awkward in how its main daily stat-tracker works, where it does not count time on a time-of-day basis, and not readily understood numbers may leave users confused. This can be fixed with time-of-day x coordinates, an aggregated half-hourly average of values for better understanding and legibility of data, and a much more psychologically pleasing 0-100 y range instead of the current 0-7.

References

- [1] Phil Thompson, Joshua Willman, and Martin Fitzpatrick. 2023. PyQt5 Reference Guide. Riverbank Computing Limited. Retrieved June 4th, 2025 from <https://www.riverbankcomputing.com/static/Docs/PyQt5/>
- [2] Gary Bradski, Vadim Pisarevsky, Anna Kogan, Shiqi Yu, and Satya Mallick. 1999. OpenCV. OpenCV. Retrieved June 4th, 2025 from <https://docs.opencv.org/4.x/>
- [3] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg and Matthias Grundmann. 2019. MediaPipe Solutions Guide. Google AI for Developers, Google AI Edge. Retrieved June 4th, 2025 from <https://ai.google.dev/edge/mediapipe/solutions/guide>
- [4] Guolin Ke. 2016. LightGBM Python-package Introduction. LightGBM 4.6.0.99 documentation. Retrieved June 4th, 2025 from <https://lightgbm.readthedocs.io/en/latest/Python-Intro.html>
- [5] David Cournapeau, Matthieu Brucher, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel. 2010. scikit-learn: machine learning in Python. scikit-learn. Retrieved June 4th, 2025 from <http://scikit-learn.org/>
- [6] Anthony Zhang and nikkie. 2014. SpeechRecognition. PyPI. Retrieved June 4th, 2025 from <https://pypi.org/project/SpeechRecognition/>
- [7] Julien Khaleghy. 2017. SerpAPI. Retrieved September 26th, 2025 from <https://serpapi.com/search-engine-apis>
- [8] Scott Lundberg, Su-In Lee. 2017. SHAP Documentation. SHAP. Retrieved October 12th, 2025 from <https://shap.readthedocs.io/en/latest/>

Appendices

A. Installation Guide (step-by-step with screenshots)

6. Clone the repo to local machine.

```
git clone https://github.com/SlyJavii/SpineWise.git
```

7. Navigate to repo directory and create a virtual environment.

```
cd SpineWise
```

```
py -m venv .venv
```

8. Activate virtual environment.

- a. MacOS and Linux only:

```
source .venv/bin/activate
```

- b. Windows:

```
.venv\Scripts\activate.bat
```

9. Install SpineWise requirements

```
py -m pip install -r requirements.txt
```

10. Launch spinewise_gui.py

```
py spinewise_gui.py
```

B. Poster (36"×48" vertical), Slides, and Video Links (Intro, Demo)

 SpineWise - Fall 2025 Capstone 2 Project Poster.pdf

 SpineWise - Presentation Slides.pdf

https://youtu.be/DJxN5FuCtqI?si=Zq_TieY40-X6mn2b (Intro)

https://youtu.be/VWRZShKFDhM?si=__E_ZB97Smdy14752 (Demo)

C. Scrum Evidence Index (links to Sprint Planning, Dailies, Reviews, Retros)

📄 Sprint Planning Meeting Minutes

📄 Daily Scrum Meeting Minutes

📄 Sprint Review Meeting Minutes

📄 Sprint Retrospective Meeting Minutes