Deep Work Guardian — Complete Project Report

Course: Artificial Intelligence — Practical (Week 1 Project) **Team:** Muhammad, Abdulla, Shabaz, Peshawa, Rasyar **Date:** February 2026

Table of Contents

---

1. Introduction

The **Deep Work Guardian** is an AI agent that runs silently in the background on a student's laptop. Its single goal is:

> **"Keep the student focused, comfortable, and productive during long study sessions."**

It does this by watching the student's environment through the webcam, microphone, battery sensor, and active window — and takes automatic actions when something goes wrong.

In simple words: Instead of the student worrying about posture, privacy, noise, battery, or distractions — the agent handles all of them automatically.

---

2. Problem Statement

Students studying for long hours on a laptop face **five common problems**:

#	Problem	What Goes Wrong
1	Sitting too close to the screen	Causes eye strain and back pain over time
2	Someone looking at your screen	Your private work, passwords, or messages get exposed
3	Noisy surroundings	You lose focus and cannot concentrate on your work
4	Laptop battery dying	Laptop shuts down or disrupts your study flow
5	Opening social media or games	Minutes turn into hours of wasted time

No single existing app solves all five problems at once. That is why we built the Deep Work Guardian — one agent that handles all five.

---

3. Proposed Solution

We split the agent into **5 independent subsystems** (one per team member). Each subsystem:

- **Watches** one specific sensor (webcam, microphone, battery, or window title)
- **Decides** if there is a problem
- **Acts** by doing something on the computer (notification, blur, play audio, change settings, close app)

All 5 subsystems run **at the same time** using Python **threads**, and they share data through a single **SharedState** object (a safe dictionary in memory).

```
Sensor (Input)  ⟶  Decision Logic  ⟶  Action (Output)
```

---

4. Agent Type and Classification

Property	Value
Agent Type	Goal-Based Agent
Goal	Maintain an optimal work environment at all times
Environment	Laptop (hardware + software) + physical workspace
Observability	Partially Observable (agent cannot see everything)
Dynamism	Dynamic (environment changes on its own)
Continuity	Continuous (not step-by-step, always running)

What does “Goal-Based” mean?

A goal-based agent has a clear goal it is trying to achieve. Every decision the agent makes is to move closer to that goal. In our case, the goal is a **distraction-free, comfortable, productive workspace**. Each subsystem checks if the current state matches the goal, and if not, it acts to fix it.

---

5. PEAS Analysis

PEAS stands for **Performance, Environment, Actuators, Sensors** — the four things you must define for any AI agent.

Performance Measures (How do we know the agent is doing well?)

- The student keeps a safe distance from the screen (≥ 50 cm)
- No stranger can see the student's screen content
- Background noise does not distract the student
- Battery does not suddenly die
- Social media or games do not waste more than 5 minutes

Environment (Where does the agent operate?)

- The laptop's operating system (Windows)
- The laptop's hardware (webcam, microphone, battery)
- The physical space around the student (desk, room, people nearby)

Actuators (What can the agent DO?)

Actuator	What it does
System Notification Popup	Shows a warning message on screen
Window Minimizer	Minimizes all open windows to hide content
Screen Blur Overlay	Creates a blurred overlay on top of the screen
OS Audio Player	Plays white noise or lo-fi music through speakers
Display Settings Controller	Changes brightness, enables dark mode
Process Manager	Closes or minimizes apps (like YouTube browser tab)

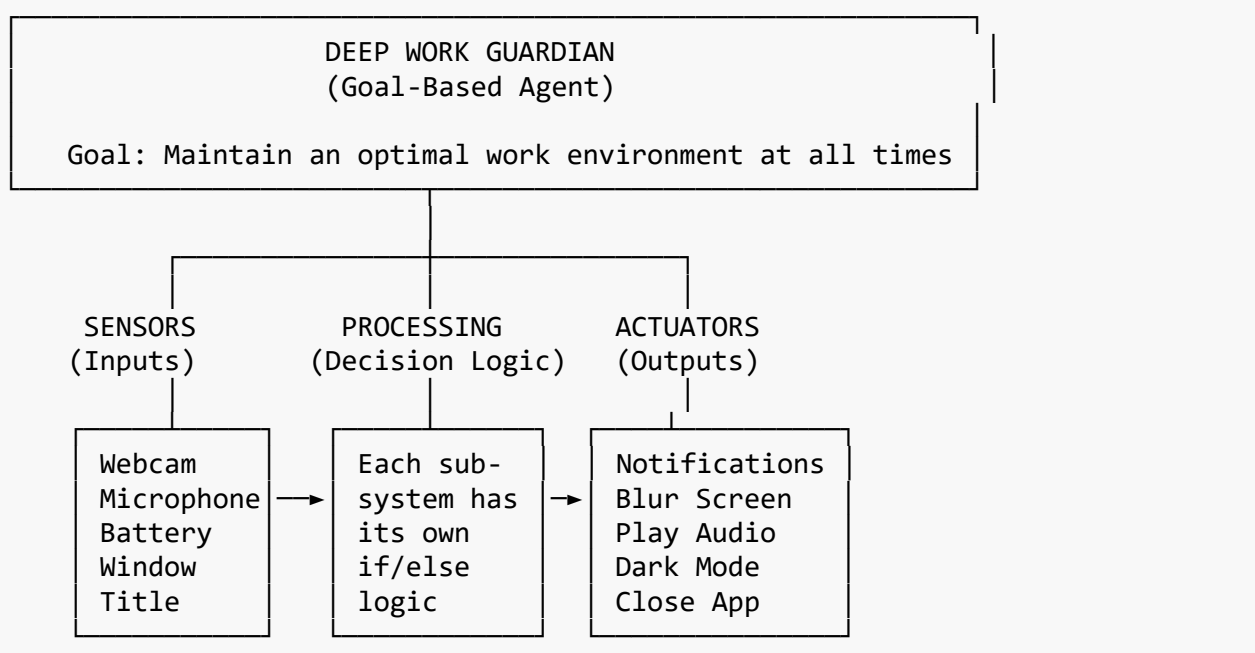
Sensors (What can the agent SEE?)

Sensor	What it reads
Webcam	The student's face (distance) + background people
Microphone	The noise level in decibels (dB) around the student
Battery Status API	Battery percentage and whether charger is plugged in
Active Window Monitor	The title of the app/window the student is using

---

6. System Architecture Overview

Below is a diagram showing how the entire system connects:



How data flows:

1. Sensors collect raw data (webcam frame, noise level, battery %, window title)
2. Each subsystem checks: “Is there a problem?” (simple if/else conditions)
3. If yes → the subsystem triggers its actuator (notification, blur, audio, etc.)
4. All subsystems run at the same time using Python threads
5. They communicate using the SharedState dictionary

---

7. GitHub Repository Structure

The project is organized on GitHub in the following folder structure:

```
ai-group-projects/
└── week-1-project/
    ├── docs/
    │   ├── REQUIREMENTS.md        ← What the system must do (this report)
    │   └── IMPLEMENTATION.md      ← How the code is written
    ├── deep-work-guardian/
    │   ├── main.py                ← Entry point — starts all threads
    │   ├── shared_state.py        ← Shared data dictionary for all agents
    │   ├── requirements.txt       ← Python packages to install
    │   └── agents/
    │       ├── __init__.py
    │       ├── ergonomics_monitor.py     ← Muhammad's subsystem
    │       ├── privacy_shield.py         ← Abdulla's subsystem
    │       ├── atmosphere_controller.py  ← Shabaz's subsystem
    │       ├── power_optimizer.py        ← Peshawa's subsystem
    │       └── distraction_blocker.py    ← Rasyar's subsystem
    └── README.md
```

What each file does:

File	Purpose
`main.py`	The file you run. It creates all agents and starts all threads.
`shared_state.py`	A thread-safe dictionary that all agents read/write to.
`ergonomics_monitor.py`	Checks face distance using webcam → sends notification.
`privacy_shield.py`	Counts faces using webcam → blurs screen if stranger found.
`atmosphere_controller.py`	Measures noise → plays white noise if too loud.
`power_optimizer.py`	Checks battery → enables dark mode if unplugged.
`distraction_blocker.py`	Checks active window → closes distracting apps after 5 min.
`requirements.txt`	List of Python libraries to install.

---

8. Shared State — How the Subsystems Communicate

Since all 5 subsystems run at the same time in separate threads, they need a safe way to share data. We use a **SharedState** class — basically a dictionary protected by a **lock** so only one thread can write at a time.

What is a Thread Lock?

When two threads try to change the same variable at the exact same time, data can get corrupted. A **lock** prevents this by making threads take turns.

SharedState Code (shared_state.py)

```
import threading

class SharedState:
    """
    A thread-safe dictionary that all 5 subsystems can read and write to.
    The lock makes sure only one thread accesses the data at a time.
    """

    def __init__(self):
        # The lock prevents two threads from writing at the same time
        self.lock = threading.Lock()

        # All shared data lives here
        self.data = {
            # --- Ergonomics Monitor (Muhammad) ---
            "face_distance_cm": 100,         # Distance of face from screen i
n cm
            "posture_warning_active": False,  # Is a warning currently showin
g?

            # --- Privacy Shield (Abdulla) ---
            "background_face_detected": False,  # Is a stranger detected?
            "screen_blurred": False,             # Is the screen currently bl
urred?

            # --- Atmosphere Controller (Shabaz) ---
            "noise_level_db": 0,              # Current noise level in decibe
ls
            "white_noise_playing": False,      # Is white noise currently pla
ying?

            # --- Power Optimizer (Peshawa) ---
            "battery_percent": 100,           # Current battery percentage
            "is_charging": True,              # Is the charger plugged in?
            "dark_mode_enabled": False,       # Is dark mode currently on?
```

```
            # --- Distraction Blocker (Rasyar) ---
            "active_window": "",              # Title of the currently active
 window
            "distraction_timer": 0,           # Seconds spent on a distractin
g app
            "app_blocked": False,             # Was an app just blocked?

            # --- Shared Webcam Frame (used by Muhammad and Abdulla) ---
            "webcam_frame": None              # The latest camera image
        }

    def get(self, key):
        """Read a value from the shared data (thread-safe)."""
        with self.lock:
            return self.data.get(key)

    def set(self, key, value):
        """Write a value to the shared data (thread-safe)."""
        with self.lock:
            self.data[key] = value

    def get_all(self):
        """Get a copy of all data (thread-safe). Used for status display."""
        with self.lock:
            return self.data.copy()
```

How it works step by step:

1. When `main.py` starts, it creates **one** SharedState object
2. This object is passed to **all 5 agents**
3. Each agent can call `shared.get("key")` to read data
4. Each agent can call `shared.set("key", value)` to write data
5. The `threading.Lock()` makes sure no two agents write at the exact same time

---

9. Subsystem 1 — Ergonomics Monitor (Muhammad)

What it does:

Watches the student through the webcam and measures how far their face is from the screen. If the student is sitting **too close** (less than 50 cm), it shows a warning notification. If the student ignores the warning, it repeats every 2 minutes.

How it works step by step:

1. Gets the latest webcam frame from SharedState

2. Uses OpenCV face detection to find the student's face
3. Calculates the distance based on how big the face appears (bigger face = closer)
4. If distance < 50 cm → shows a notification popup
5. Waits 2 minutes before showing the next warning (to avoid spamming)
6. Repeats every 1 second

Sensor and Actuator:

Part	Detail
Sensor	Webcam (reads face position and size)
Actuator	System notification popup ("You're too close!")
Threshold	50 cm — if face is closer than this, warn the user
Repeat	Warning repeats every 2 minutes if problem persists
Privacy	No webcam footage is ever saved or recorded

Code (ergonomics_monitor.py):

```
import cv2
import time
from plyer import notification

class ErgonomicsMonitor:
    """
    Muhammad's subsystem.
    Monitors how far the student's face is from the screen.
    Sends a notification if the student is sitting too close.
    """

    def __init__(self, shared_state):
        self.shared = shared_state
        # Load OpenCV's built-in face detection model
        self.face_cascade = cv2.CascadeClassifier(
            cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'
        )
        # Track when the last warning was sent
        self.last_warning_time = 0
        # Known face width in cm (average human face is ~14 cm wide)
        self.KNOWN_FACE_WIDTH_CM = 14.0
        # Focal length (calibrated for common webcams, adjust if needed)
        self.FOCAL_LENGTH = 500

    def run(self):
        """Main loop — runs forever in its own thread."""
        while True:
            # Step 1: Get the latest webcam frame from shared state
            frame = self.shared.get("webcam_frame")
```

```
            if frame is not None:
                # Step 2: Calculate face distance
                distance = self.calculate_face_distance(frame)
                self.shared.set("face_distance_cm", distance)

                # Step 3: If too close, warn the user
                if distance < 50:
                    self.warn_user_if_needed()
                else:
                    self.shared.set("posture_warning_active", False)

            # Check every 1 second
            time.sleep(1)

    def calculate_face_distance(self, frame):
        """
        Detects the user's face and estimates distance from the screen.
        Uses the formula: distance = (known_width × focal_length) / pixel_wid
th
        """
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = self.face_cascade.detectMultiScale(gray, 1.3, 5)

        if len(faces) > 0:
            # Take the first (largest) face — this is the user
            x, y, w, h = faces[0]
            # Calculate distance using the face width in pixels
            distance = (self.KNOWN_FACE_WIDTH_CM * self.FOCAL_LENGTH) / w
            return round(distance, 1)

        # If no face detected, assume safe distance
        return 100

    def warn_user_if_needed(self):
        """Shows a warning notification, but only once every 2 minutes."""
        current_time = time.time()

        # Only warn if 2 minutes (120 seconds) have passed since last warning
        if current_time - self.last_warning_time > 120:
            self.last_warning_time = current_time
            self.shared.set("posture_warning_active", True)

            # Show desktop notification
            notification.notify(
                title="⚠ Posture Warning",
                message="You are sitting too close to the screen! Please move
 back.",
                timeout=10
            )
```

The Distance Formula Explained:

```
distance = (known_face_width × focal_length) / face_width_in_pixels
```

- **known_face_width**: Average human face is about 14 cm wide
- **focal_length**: A camera constant (we use 500, which works for most webcams)
- **face_width_in_pixels**: How many pixels wide the face is in the image

If the face looks **big** in the image (many pixels) → the person is **close**. If the face looks **small** (few pixels) → the person is **far away**.

---

10. Subsystem 2 — Privacy Shield (Abdulla)

What it does:

Uses the same webcam feed to scan for **additional faces** in the background. If a second person is detected (someone looking over the student's shoulder), it immediately **minimizes all windows** or **blurs the screen** to protect privacy. When the person leaves, it automatically restores the screen.

How it works step by step:

1. Gets the latest webcam frame from SharedState (same camera as Subsystem 1)
2. Uses OpenCV face detection to count all faces in the image
3. If more than 1 face → someone else is looking → blur/minimize the screen
4. If only 1 face (or none) → restore the screen to normal
5. Repeats every 0.5 seconds (faster than Subsystem 1 because privacy is urgent)

Sensor and Actuator:

Part	Detail
Sensor	Webcam (counts number of faces in the frame)
Actuator	Minimizes all windows OR creates a blur overlay on screen
Trigger	More than 1 face detected
Recovery	Automatically restores when the extra face disappears

Code (privacy_shield.py):

```
import cv2
import time
import pygetwindow as gw

class PrivacyShield:
    """
    Abdulla's subsystem.
    Detects if someone is looking over the student's shoulder.
    If a second face is found, minimizes all windows to protect privacy.
    """

    def __init__(self, shared_state):
        self.shared = shared_state
        # Load face detection model
        self.face_cascade = cv2.CascadeClassifier(
            cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'
        )
        # Keep track of window positions before minimizing
        self.windows_were_minimized = False

    def run(self):
        """Main loop — runs forever in its own thread."""
        while True:
            frame = self.shared.get("webcam_frame")

            if frame is not None:
                # Count all faces in the frame
                num_faces = self.count_faces(frame)

                if num_faces > 1:
                    # Stranger detected! Protect privacy
                    self.shared.set("background_face_detected", True)
                    if not self.windows_were_minimized:
                        self.minimize_all_windows()
                else:
                    # No stranger — restore if needed
                    self.shared.set("background_face_detected", False)
                    if self.windows_were_minimized:
                        self.restore_windows()

            # Check every 0.5 seconds (privacy is urgent)
            time.sleep(0.5)

    def count_faces(self, frame):
        """Detects all faces in the webcam frame and returns the count."""
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = self.face_cascade.detectMultiScale(gray, 1.3, 5)
```

```
        return len(faces)

    def minimize_all_windows(self):
        """Minimizes all open windows to hide sensitive content."""
        try:
            windows = gw.getAllWindows()
            for window in windows:
                if window.title and not window.isMinimized:
                    window.minimize()
            self.windows_were_minimized = True
            self.shared.set("screen_blurred", True)
            print("[Privacy Shield] ⚠ Stranger detected! Windows minimized.
")
        except Exception as e:
            print(f"[Privacy Shield] Error minimizing windows: {e}")

    def restore_windows(self):
        """Restores all previously minimized windows."""
        try:
            windows = gw.getAllWindows()
            for window in windows:
                if window.title and window.isMinimized:
                    window.restore()
            self.windows_were_minimized = False
            self.shared.set("screen_blurred", False)
            print("[Privacy Shield] ☑ Stranger gone. Windows restored.")
        except Exception as e:
            print(f"[Privacy Shield] Error restoring windows: {e}")
```

Important: Webcam Sharing Between Subsystems 1 and 2

Both Muhammad (Ergonomics) and Abdulla (Privacy Shield) need the webcam. Instead of opening two separate cameras, we use **one shared webcam thread** in `main.py` that captures frames and puts them in SharedState. Both subsystems read from `shared.get("webcam_frame")`.

---

11. Subsystem 3 — Atmosphere Controller (Shabaz)

What it does:

Listens through the laptop's microphone and measures how loud the environment is in **decibels (dB)**. If the noise level goes above the threshold (default: 60 dB), it automatically plays **white noise or lo-fi music** to help the student block out the noise. When the environment becomes quiet again, the audio stops automatically.

How it works step by step:

1. Opens the microphone using PyAudio
2. Reads a small chunk of audio data
3. Calculates the RMS (Root Mean Square) of the audio — this gives the loudness
4. Converts RMS to decibels (dB)
5. If dB > 60 → plays white noise through the speakers
6. If dB < 60 → stops the white noise
7. Repeats every 1 second

Sensor and Actuator:

Part	Detail
Sensor	Microphone (measures ambient noise in decibels)
Actuator	OS audio player (plays white noise or lo-fi music)
Threshold	60 dB — above this is considered distracting noise
Recovery	Audio stops automatically when noise drops below 60 dB

Code (atmosphere_controller.py):

```
import pyaudio
import numpy as np
import time
import threading

class AtmosphereController:
    """
    Shabaz's subsystem.
    Monitors background noise through the microphone.
    Plays white noise when the environment is too loud.
    """

    def __init__(self, shared_state):
        self.shared = shared_state
        self.threshold_db = 60       # Noise threshold in decibels
        self.audio_interface = None   # PyAudio instance
        self.mic_stream = None        # Microphone stream
        self.is_playing = False       # Is white noise currently playing?

        # Audio settings
        self.CHUNK = 1024             # Number of audio samples per read
        self.RATE = 44100             # Samples per second (CD quality)
        self.FORMAT = pyaudio.paInt16 # 16-bit audio

    def run(self):
        """Main loop — runs forever in its own thread."""
        # Step 1: Open the microphone
```

```
        self.start_microphone()

        while True:
            # Step 2: Measure current noise level
            db_level = self.measure_noise_level()
            self.shared.set("noise_level_db", round(db_level, 1))

            # Step 3: If too loud, play white noise
            if db_level > self.threshold_db:
                if not self.is_playing:
                    self.play_white_noise()
            else:
                # Environment is quiet — stop white noise
                if self.is_playing:
                    self.stop_white_noise()

            time.sleep(1)

    def start_microphone(self):
        """Opens the microphone for reading audio."""
        try:
            self.audio_interface = pyaudio.PyAudio()
            self.mic_stream = self.audio_interface.open(
                format=self.FORMAT,
                channels=1,            # Mono audio
                rate=self.RATE,
                input=True,            # We are reading (not writing) audio
                frames_per_buffer=self.CHUNK
            )
            print("[Atmosphere] 🎤 Microphone opened successfully.")
        except Exception as e:
            print(f"[Atmosphere] Error opening microphone: {e}")

    def measure_noise_level(self):
        """
        Reads audio from mic and calculates the noise level in decibels.
        Formula: dB = 20 × log10(RMS)
        """
        try:
            # Read raw audio data from microphone
            raw_data = self.mic_stream.read(self.CHUNK, exception_on_overflow
=False)

            # Convert to numpy array of integers
            audio_array = np.frombuffer(raw_data, dtype=np.int16)

            # Calculate RMS (Root Mean Square) — represents average loudness
            rms = np.sqrt(np.mean(audio_array.astype(float) ** 2))
```

```
            # Convert RMS to decibels
            if rms > 0:
                db = 20 * np.log10(rms)
            else:
                db = 0

            return db
        except Exception:
            return 0

    def play_white_noise(self):
        """Starts playing white noise audio."""
        self.is_playing = True
        self.shared.set("white_noise_playing", True)
        print("[Atmosphere] 🔊 Too loud! Playing white noise...")

        # Generate and play white noise in a separate thread
        def _generate_and_play():
            try:
                output_stream = self.audio_interface.open(
                    format=pyaudio.paFloat32,
                    channels=1,
                    rate=self.RATE,
                    output=True
                )
                while self.is_playing:
                    # Generate random noise (white noise)
                    noise = np.random.uniform(-0.1, 0.1, self.CHUNK).astype(n
p.float32)
                    output_stream.write(noise.tobytes())
                output_stream.close()
            except Exception as e:
                print(f"[Atmosphere] Error playing white noise: {e}")

        threading.Thread(target=_generate_and_play, daemon=True).start()

    def stop_white_noise(self):
        """Stops the white noise audio."""
        self.is_playing = False
        self.shared.set("white_noise_playing", False)
        print("[Atmosphere] 🔇 Environment is quiet. White noise stopped.")
```

The Decibel Formula Explained:

$$dB = 20 \times \log_{10}(RMS)$$

- **RMS** (Root Mean Square) is the average loudness of the audio signal
- **$\log_{10}$** converts the raw number to a human-friendly scale
- **20×** is the standard multiplier for sound pressure levels

dB Level	What it sounds like
20 dB	Whisper
40 dB	Quiet library
60 dB	Normal conversation
80 dB	Busy street
100 dB	Concert / shouting

---

12. Subsystem 4 — Power Optimizer (Peshawa)

What it does:

Monitors the laptop's battery and charging status. When the charger is **unplugged**, it automatically: - Switches the OS to **Dark Mode** (saves battery on modern screens) - **Lowers the screen brightness** (the biggest power consumer)

When the charger is **plugged back in**, it automatically restores all settings to normal.

How it works step by step:

1. Uses `psutil` library to read battery percentage and charging status
2. If charger is unplugged → enable dark mode + lower brightness
3. If charger is plugged in → disable dark mode + restore brightness
4. Repeats every 5 seconds (battery status doesn't change rapidly)

Sensor and Actuator:

Part	Detail
Sensor	Battery/Charger Status API (via psutil)
Actuator	Dark mode toggle + brightness control
Trigger	Charger is unplugged
Recovery	Automatically restores when charger is plugged in

Code (power_optimizer.py):

```
import psutil
import subprocess
import time

class PowerOptimizer:
    """
    Peshawa's subsystem.
    Monitors battery status and saves power when the charger is unplugged.
    Enables dark mode and lowers brightness automatically.
```

```
    """

    def __init__(self, shared_state):
        self.shared = shared_state
        self.power_saving_active = False

    def run(self):
        """Main loop — runs forever in its own thread."""
        while True:
            # Step 1: Read battery info
            battery = psutil.sensors_battery()

            if battery is not None:
                percent = battery.percent
                charging = battery.power_plugged

                # Update shared state
                self.shared.set("battery_percent", percent)
                self.shared.set("is_charging", charging)

                # Step 2: If unplugged, save power
                if not charging and not self.power_saving_active:
                    self.enable_power_saving()

                # Step 3: If plugged in, restore normal
                elif charging and self.power_saving_active:
                    self.restore_normal_mode()

            # Check every 5 seconds
            time.sleep(5)

    def enable_power_saving(self):
        """Turns on dark mode and lowers brightness to save battery."""
        self.power_saving_active = True
        self.shared.set("dark_mode_enabled", True)
        print("[Power] 🔋 Charger unplugged! Enabling power saving mode...")

        # Enable Windows Dark Mode via Registry
        self.set_dark_mode(True)

        # Lower brightness to 30%
        self.set_brightness(30)

    def restore_normal_mode(self):
        """Restores normal mode when charger is plugged back in."""
        self.power_saving_active = False
        self.shared.set("dark_mode_enabled", False)
        print("[Power] ⚡ Charger connected! Restoring normal mode...")
```

```
        # Disable Windows Dark Mode
        self.set_dark_mode(False)

        # Restore brightness to 80%
        self.set_brightness(80)

    def set_dark_mode(self, enable):
        """Toggles Windows Dark Mode using a PowerShell command."""
        try:
            # 0 = dark mode ON, 1 = dark mode OFF
            value = 0 if enable else 1
            command = (
                f'Set-ItemProperty -Path '
                f'HKCU:\\Software\\Microsoft\\Windows\\CurrentVersion\\Themes
\\Personalize '
                f'-Name AppsUseLightTheme -Value {value}'
            )
            subprocess.run(["powershell", "-Command", command], capture_outpu
t=True)
        except Exception as e:
            print(f"[Power] Error setting dark mode: {e}")

    def set_brightness(self, level):
        """Sets screen brightness using PowerShell (Windows only)."""
        try:
            command = (
                f'(Get-WmiObject -Namespace root/WMI '
                f'-Class WmiMonitorBrightnessMethods)'
                f'.WmiSetBrightness(1, {level})'
            )
            subprocess.run(["powershell", "-Command", command], capture_outpu
t=True)
        except Exception as e:
            print(f"[Power] Error setting brightness: {e}")
```

---

13. Subsystem 5 — Distraction Blocker (Rasyar)

What it does:

Checks the title of the window the student is currently using every 2 seconds. If the student opens a **distracting app** (YouTube, Instagram, Facebook, TikTok, games, etc.), it starts a **5-minute timer**. If the student stays on the distracting app for more than 5 continuous minutes, it **minimizes the window** and sends a notification.

How it works step by step:

1. Uses `pygetwindow` to read the title of the currently active window
2. Compares the title against a blocklist of distracting keywords
3. If the window is distracting → starts a 5-minute timer
4. If the student switches away before 5 minutes → timer resets
5. If 5 minutes pass on the distracting app → minimizes the window and sends a warning
6. Repeats every 2 seconds

Sensor and Actuator:

Part	Detail
Sensor	Active window title (via pygetwindow)
Actuator	Minimizes/closes the distracting window + notification
Blocklist	youtube, instagram, facebook, twitter, tiktok, game
Timer	5 minutes (300 seconds) of continuous use before blocking

Code (distraction_blocker.py):

```
import time
import pygetwindow as gw
from plyer import notification

class DistractionBlocker:
    """
    Rasyar's subsystem.
    Monitors which app the student is using.
    If the student spends more than 5 minutes on a distracting app,
    it minimizes the app and sends a warning.
    """

    def __init__(self, shared_state):
        self.shared = shared_state
        # List of keywords that indicate distracting apps
        self.blocklist = [
            "youtube", "instagram", "facebook",
            "twitter", "tiktok", "game", "netflix",
            "reddit", "twitch"
        ]
        # Timer tracking
        self.distraction_start_time = None
        self.timeout_seconds = 300  # 5 minutes = 300 seconds

    def run(self):
        """Main loop — runs forever in its own thread."""
        while True:
```

```
            # Step 1: Get the title of the currently active window
            active_window = self.get_active_window_title()
            self.shared.set("active_window", active_window)

            # Step 2: Check if this window is distracting
            if self.is_distraction(active_window):
                self.handle_distraction(active_window)
            else:
                # Not distracting — reset the timer
                self.distraction_start_time = None
                self.shared.set("distraction_timer", 0)
                self.shared.set("app_blocked", False)

            # Check every 2 seconds
            time.sleep(2)

    def get_active_window_title(self):
        """Returns the title of the currently focused window."""
        try:
            window = gw.getActiveWindow()
            if window and window.title:
                return window.title
            return ""
        except Exception:
            return ""

    def is_distraction(self, window_title):
        """Checks if the window title contains any blocked keyword."""
        title_lower = window_title.lower()
        return any(keyword in title_lower for keyword in self.blocklist)

    def handle_distraction(self, window_title):
        """
        Manages the distraction timer.
        If the student has been on a distracting app for > 5 minutes,
        it minimizes the window and sends a notification.
        """
        # Start the timer if this is the first detection
        if self.distraction_start_time is None:
            self.distraction_start_time = time.time()

        # Calculate how long the student has been distracted
        elapsed = time.time() - self.distraction_start_time
        self.shared.set("distraction_timer", round(elapsed))

        # If over 5 minutes, block the app
        if elapsed > self.timeout_seconds:
            self.block_app(window_title)
```

```
    def block_app(self, window_title):
        """Minimizes the distracting window and sends a notification."""
        try:
            # Find and minimize the distracting window
            window = gw.getActiveWindow()
            if window:
                window.minimize()

            self.shared.set("app_blocked", True)

            # Send notification
            notification.notify(
                title="🚫 Distraction Blocked",
                message=f"You spent over 5 minutes on '{window_title}'. "
                        f"Time to get back to work!",
                timeout=10
            )

            # Reset timer
            self.distraction_start_time = None
            print(f"[Distraction] 🚫 Blocked: {window_title}")

        except Exception as e:
            print(f"[Distraction] Error blocking app: {e}")
```

---

14. Main Runner — How Everything Starts

The `main.py` file is the entry point. When you run it, it:

1. Creates the SharedState (the shared dictionary)
2. Creates all 5 subsystem objects
3. Starts a webcam thread (shared by Muhammad and Abdulla)
4. Starts 5 agent threads (one per subsystem)
5. Prints a live status dashboard every 5 seconds

Code (main.py):

```
import threading
import time
import cv2
from shared_state import SharedState
from agents.ergonomics_monitor import ErgonomicsMonitor
from agents.privacy_shield import PrivacyShield
from agents.atmosphere_controller import AtmosphereController
from agents.power_optimizer import PowerOptimizer
from agents.distraction_blocker import DistractionBlocker
```

```python
def webcam_capture_thread(shared_state):
    """
    Captures webcam frames continuously and stores them in shared state.
    This thread is shared by both the Ergonomics Monitor (Muhammad)
    and the Privacy Shield (Abdulla).
    """
    cap = cv2.VideoCapture(0)

    if not cap.isOpened():
        print("[Webcam] ❌ Could not open webcam!")
        return

    print("[Webcam] 📷 Webcam opened successfully.")

    while True:
        ret, frame = cap.read()
        if ret:
            shared_state.set("webcam_frame", frame)
        time.sleep(0.1)  # Capture at ~10 FPS

    cap.release()


def main():
    print("=" * 55)
    print("   🌓  DEEP WORK GUARDIAN — Starting Up...")
    print("=" * 55)
    print()

    # Step 1: Create the shared state
    shared = SharedState()

    # Step 2: Create all 5 agent objects
    ergonomics   = ErgonomicsMonitor(shared)      # Muhammad
    privacy      = PrivacyShield(shared)           # Abdulla
    atmosphere   = AtmosphereController(shared)    # Shabaz
    power        = PowerOptimizer(shared)          # Peshawa
    distraction  = DistractionBlocker(shared)      # Rasyar

    # Step 3: Start the webcam thread (shared by Muhammad and Abdulla)
    webcam_thread = threading.Thread(
        target=webcam_capture_thread,
        args=(shared,),
        daemon=True
    )
    webcam_thread.start()
```

```
    # Give the webcam a moment to warm up
    time.sleep(1)

    # Step 4: Start all 5 agent threads
    threads = [
        threading.Thread(target=ergonomics.run,  daemon=True, name="Ergonomic
s"),
        threading.Thread(target=privacy.run,     daemon=True, name="Privacy
"),
        threading.Thread(target=atmosphere.run,  daemon=True, name="Atmospher
e"),
        threading.Thread(target=power.run,       daemon=True, name="Power"),
        threading.Thread(target=distraction.run, daemon=True, name="Distracti
on"),
    ]

    for t in threads:
        t.start()
        print(f"  ☑ {t.name} subsystem started.")

    print()
    print("All subsystems are now running!")
    print("Press Ctrl+C to stop the guardian.\n")

    # Step 5: Keep main thread alive and print status
    try:
        while True:
            status = shared.get_all()
            print(
                f"  📏 Distance: {status['face_distance_cm']}cm  |  "
                f"👤 Stranger: {'YES' if status['background_face_detected']
else 'No'}  |  "
                f"🔊 Noise: {status['noise_level_db']}dB  |  "
                f"🔋 Battery: {status['battery_percent']}%  |  "
                f"🪟 Window: {status['active_window'][:25]}"
            )
            time.sleep(5)
    except KeyboardInterrupt:
        print("\n👋 Deep Work Guardian shutting down. Goodbye!")


if __name__ == "__main__":
    main()
```

---

15. How Python Threads Work in This Project

What is a Thread?

A **thread** is like a lightweight worker inside your program. Normally, Python runs one line of code after another (sequentially). With threads, you can have **multiple workers running at the same time**.

Why do we need Threads?

Our 5 subsystems need to run **simultaneously** — not one after the other. For example: - While the Ergonomics Monitor is checking face distance... - ...the Atmosphere Controller should be checking noise level... - ...and the Distraction Blocker should be scanning the active window.

Without threads, we would have to wait for each subsystem to finish before the next one starts, which would make the agent extremely slow.

How Threads Work in Our Code:

```
# 1. Import threading
import threading

# 2. Create a thread that runs a function
thread = threading.Thread(target=some_function, daemon=True)

# 3. Start the thread (it now runs in the background)
thread.start()
```

- `target=some_function` — the function that this thread will run
- `daemon=True` — the thread will automatically stop when the main program exits
- `.start()` — tells the thread to begin running

Visual Diagram of Thread Execution:

```
Time ──────────────────────────────────────────────────────────►

Main Thread:      [Start] → [Create Agents] → [Print Status every 5s] → ...

Webcam Thread:    ..................[Capture Frame]→[Capture]→[Capture]→...

Thread 1 (Muhammad):  ........[Check Distance]→[Check]→[Check]→...

Thread 2 (Abdulla):   ........[Count Faces]→[Count]→[Count]→...
```

```
Thread 3 (Shabaz):    ........[Measure Noise]→[Measure]→[Measure]→...

Thread 4 (Peshawa):   ........[Check Battery]→[Check]→[Check]→...

Thread 5 (Rasyar):    ........[Check Window]→[Check]→[Check]→...
```

All threads run **at the same time** — the operating system switches between them very quickly.

Why SharedState needs a Lock:

Since all threads access the same `data` dictionary, we use a `threading.Lock()` to prevent corruption:

```
# WITHOUT a lock (DANGEROUS):
# Thread 1: data["x"] = 10  ←── these two could happen at
# Thread 2: data["x"] = 20  ←── the exact same time = BUG

# WITH a lock (SAFE):
# Thread 1: acquire lock → data["x"] = 10 → release lock
# Thread 2: wait... → acquire lock → data["x"] = 20 → release lock
```

---

16. Required Libraries

These are the Python packages you need to install:

Package	What it does	Used by
`opencv-python`	Webcam access and face detection	Muhammad, Abdulla
`pyaudio`	Microphone input for noise measurement	Shabaz
`numpy`	Math operations (RMS calculation)	Shabaz
`psutil`	Battery status and process management	Peshawa, Rasyar
`pygetwindow`	Read/control active windows on Windows	Abdulla, Rasyar
`plyer`	Desktop notification popups	Muhammad, Rasyar

requirements.txt:

```
opencv-python
pyaudio
numpy
psutil
PyGetWindow
plyer
```

---

17. How to Install and Run

Step 1: Clone the Repository

```
git clone https://github.com/rasathelamedude/ai-group-projects.git
cd ai-group-projects/week-1-project/deep-work-guardian
```

Step 2: Install Dependencies

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

> **Note about PyAudio on Windows:** If `pip install pyaudio` fails, download the `.whl` file from https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyaudio and install with `pip install <filename>.whl`.

Step 3: Run the Agent

```
python main.py
```

Step 4: Stop the Agent

Press `Ctrl+C` in the terminal.

---

18. Summary of Every Member's Work

Member	Subsystem	File	What They Built
Muhammad	Ergonomics Monitor	`ergonomics_monitor.py`	Reads the webcam, detects the user's face, calculates distance. Shows a notification if < 50 cm.
Abdulla	Privacy Shield	`privacy_shield.py`	Reads the webcam, counts faces. If > 1 face, minimizes all windows. Restores when stranger leaves.
Shabaz	Atmosphere Controller	`atmosphere_controller.py`	Reads the microphone, measures noise in dB. If > 60 dB, plays white noise. Stops when quiet.
Peshawa	Power Optimizer	`power_optimizer.py`	Reads battery status. If unplugged, enables dark mode + lowers brightness. Restores when plugged.
Rasyar	Distraction Blocker	`distraction_blocker.py`	Reads active window title. If distracting app > 5 min, minimizes it and sends a warning.

Shared Work (All Members):

- **SharedState** (`shared_state.py`) — The thread-safe dictionary all agents use
- **Main Runner** (`main.py`) — The script that starts everything with threads
- **Integration Testing** — Making sure all 5 subsystems run together without crashing

---

Deep Work Guardian — AI Practical Course, Week 1 Project, February 2026