# Student Performance System — Full Code Export
# 2025-10-14 13:08

## Admin Dashboard (admin.py)

```
#!/usr/bin/env python3
"""
Admin Dashboard for Student Performance Monitoring System
"""

import tkinter as tk
from tkinter import ttk, messagebox
from database import db
import datetime
import sys
import os
import subprocess

class AdminDashboard:
    def __init__(self, user):
        self.user = user
        self.root = tk.Tk()
        self.root.title("Admin Dashboard - Student Performance Monitoring System")
        self.root.geometry("1400x800")
        self.root.state('zoomed')  # Maximize window

        # Configure grid weights
        self.root.grid_columnconfigure(1, weight=1)
        self.root.grid_rowconfigure(0, weight=1)

        # Create sidebar
        self.create_sidebar()

        # Create main content area
        self.create_main_content()

        # Load initial data
        self.load_dashboard_data()
        self.load_students()
        self.load_teachers()

        # Handle window close
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)

    def create_sidebar(self):
        """Create left sidebar with navigation"""
        sidebar = tk.Frame(self.root, bg="white", width=250)
        sidebar.grid(row=0, column=0, sticky="nsew", padx=0, pady=0)
        sidebar.grid_propagate(False)

        # Time and date
        time_frame = tk.Frame(sidebar, bg="white")
        time_frame.pack(fill="x", padx=20, pady=20)

        current_time = datetime.datetime.now().strftime("%H:%M:%S")
        current_date = datetime.datetime.now().strftime("%Y/%m/%d")

        time_label = tk.Label(time_frame, text=f"■ {current_time}",
                              font=("Arial", 12, "bold"), fg="#2c3e50", bg="white")
        time_label.pack(anchor="w")

        date_label = tk.Label(time_frame, text=current_date,
                              font=("Arial", 10), fg="#7f8c8d", bg="white")
        date_label.pack(anchor="w", pady=(5, 0))

        # Profile section
        profile_frame = tk.Frame(sidebar, bg="white")
        profile_frame.pack(fill="x", padx=20, pady=20)

        # Profile picture (simple circle)
        profile_canvas = tk.Canvas(profile_frame, width=60, height=60, bg="white", highlightthickness=0)
        profile_canvas.pack(pady=10)
        profile_canvas.create_oval(5, 5, 55, 55, fill="#3498db", outline="")
        profile_canvas.create_text(30, 30, text="■■■", font=("Arial", 20), fill="white")

        # Admin name
        admin_name = tk.Label(profile_frame, text="System Administrator",
                              font=("Arial", 12, "bold"), fg="#2c3e50", bg="white")
        admin_name.pack()
        # Navigation menu
```

```
        nav_frame = tk.Frame(sidebar, bg="white")
        nav_frame.pack(fill="x", padx=20, pady=20)

        # Menu items
        menu_items = [
            ("■ Dashboard", self.show_dashboard),
            ("■ Manage Students", self.show_students),
            ("■■■ Manage Teachers", self.show_teachers),
            ("■ Predictions", self.show_predictions),
            ("■■ Settings", self.show_settings),
            ("■ Exit", self.logout)
        ]

        self.menu_buttons = {}
        for text, command in menu_items:
            btn = tk.Button(nav_frame, text=text, font=("Arial", 11),
                            fg="#2c3e50", bg="white", relief="flat",
                            anchor="w", command=command, cursor="hand2",
                            activebackground="#ecf0f1", activeforeground="#2c3e50")
            btn.pack(fill="x", pady=2)
            self.menu_buttons[text] = btn

        # Highlight dashboard as active
        self.highlight_menu("■ Dashboard")

    def create_main_content(self):
        """Create main content area"""
        self.main_frame = tk.Frame(self.root, bg="#f8f9fa")
        self.main_frame.grid(row=0, column=1, sticky="nsew", padx=0, pady=0)

        # Top header
        self.create_header()

        # Create scrollable content area
        self.create_scrollable_content()

        # Configure the scrollable frame for grid layout
        self.scrollable_frame.grid_columnconfigure(0, weight=1)
        self.scrollable_frame.grid_rowconfigure(0, weight=1)

        # Create different content pages
        self.create_dashboard_content()
        self.create_students_content()
        self.create_teachers_content()
        self.create_predictions_content()
        self.create_settings_content()

        # Show dashboard by default
        self.show_dashboard()

    def create_header(self):
        """Create top header bar"""
        header = tk.Frame(self.main_frame, bg="#3498db", height=60)
        header.pack(fill="x", side="top")
        header.pack_propagate(False)

        # Title
        title_label = tk.Label(header, text="System Management Dashboard",
                               font=("Arial", 18, "bold"), fg="white", bg="#3498db")
        title_label.pack(side="left", padx=20, pady=15)

        # Logout button
        logout_btn = tk.Button(header, text="Logout", font=("Arial", 11, "bold"),
                               fg="white", bg="#27ae60", relief="flat",
                               cursor="hand2", command=self.logout,
                               activebackground="#229954", activeforeground="white")
        logout_btn.pack(side="right", padx=20, pady=15)

    def create_scrollable_content(self):
        """Create scrollable content area"""
        # Create canvas and scrollbar for scrolling
        self.canvas = tk.Canvas(self.main_frame, bg="#f8f9fa", highlightthickness=0)
        self.scrollbar = ttk.Scrollbar(self.main_frame, orient="vertical", command=self.canvas.yview)
        self.scrollable_frame = tk.Frame(self.canvas, bg="#f8f9fa")

        # Configure scrolling
        self.scrollable_frame.bind(
            "<Configure>",
            lambda e: self.canvas.configure(scrollregion=self.canvas.bbox("all"))
        )

        self.canvas.create_window((0, 0), window=self.scrollable_frame, anchor="nw")
        self.canvas.configure(yscrollcommand=self.scrollbar.set)

        # Pack canvas and scrollbar
        self.canvas.pack(side="left", fill="both", expand=True)
```

```
        self.scrollbar.pack(side="right", fill="y")

        # Create a container inside the scrollable frame that will hold different pages
        self.content_frame = tk.Frame(self.scrollable_frame, bg="#f8f9fa")
        self.content_frame.grid(row=0, column=0, sticky="nsew")
        self.scrollable_frame.grid_columnconfigure(0, weight=1)
        self.scrollable_frame.grid_rowconfigure(0, weight=1)

        # Optional: bind mouse wheel for smoother scrolling
        self.canvas.bind_all("<MouseWheel>", lambda e: self.canvas.yview_scroll(int(-1*(e.delta/120)), "unit

    def create_dashboard_content(self):
        """Create dashboard content and sections"""
        self.dashboard_content = tk.Frame(self.content_frame, bg="#f8f9fa")
        # Grid weights for dashboard
        self.dashboard_content.grid_columnconfigure(0, weight=1)
        self.dashboard_content.grid_columnconfigure(1, weight=1)
        self.dashboard_content.grid_columnconfigure(2, weight=1)
        self.dashboard_content.grid_rowconfigure(0, weight=0)  # could be future title
        self.dashboard_content.grid_rowconfigure(1, weight=0)  # stats cards
        self.dashboard_content.grid_rowconfigure(2, weight=1)  # charts
        self.dashboard_content.grid_rowconfigure(3, weight=0)  # top students title
        self.dashboard_content.grid_rowconfigure(4, weight=0)  # top students cards

        # Build the dashboard sections
        self.create_stats_cards()
        self.create_charts_section()
        self.create_top_students_section()

    def run(self):
        """Start the Tkinter main loop"""
        self.root.mainloop()

    def create_stats_cards(self):
        """Create statistics cards using grid layout"""
        # Card 1: Total Students
        card1 = tk.Frame(self.dashboard_content, bg="#1abc9c", relief="flat", bd=0)
        card1.grid(row=1, column=0, padx=10, pady=10, sticky="nsew")
        card1.grid_columnconfigure(0, weight=1)
        card1.grid_rowconfigure(0, weight=1)

        tk.Label(card1, text="Total Students", font=("Arial", 12, "bold"),
                fg="white", bg="#1abc9c").grid(row=0, column=0, sticky="w", padx=15, pady=(15, 5))

        # Icon
        icon_frame = tk.Frame(card1, bg="#1abc9c")
        icon_frame.grid(row=1, column=0, sticky="w", padx=15, pady=5)
        tk.Label(icon_frame, text="■", font=("Arial", 20), fg="white", bg="#1abc9c").grid(row=0, column=0)

        self.students_count_label = tk.Label(card1, text="0", font=("Arial", 24, "bold"),
                                             fg="white", bg="#1abc9c")
        self.students_count_label.grid(row=2, column=0, sticky="w", padx=15, pady=(0, 15))

        # Card 2: Total Teachers
        card2 = tk.Frame(self.dashboard_content, bg="#e74c3c", relief="flat", bd=0)
        card2.grid(row=1, column=1, padx=10, pady=10, sticky="nsew")
        card2.grid_columnconfigure(0, weight=1)
        card2.grid_rowconfigure(0, weight=1)

        tk.Label(card2, text="Total Teachers", font=("Arial", 12, "bold"),
                fg="white", bg="#e74c3c").grid(row=0, column=0, sticky="w", padx=15, pady=(15, 5))

        icon_frame2 = tk.Frame(card2, bg="#e74c3c")
        icon_frame2.grid(row=1, column=0, sticky="w", padx=15, pady=5)
        tk.Label(icon_frame2, text="■■■", font=("Arial", 20), fg="white", bg="#e74c3c").grid(row=0, column=0

        self.teachers_count_label = tk.Label(card2, text="0", font=("Arial", 24, "bold"),
                                             fg="white", bg="#e74c3c")
        self.teachers_count_label.grid(row=2, column=0, sticky="w", padx=15, pady=(0, 15))

        # Card 3: Average Performance
        card3 = tk.Frame(self.dashboard_content, bg="#f39c12", relief="flat", bd=0)
        card3.grid(row=1, column=2, padx=10, pady=10, sticky="nsew")
        card3.grid_columnconfigure(0, weight=1)
        card3.grid_rowconfigure(0, weight=1)

        tk.Label(card3, text="Avg Performance", font=("Arial", 12, "bold"),
                fg="white", bg="#f39c12").grid(row=0, column=0, sticky="w", padx=15, pady=(15, 5))

        icon_frame3 = tk.Frame(card3, bg="#f39c12")
        icon_frame3.grid(row=1, column=0, sticky="w", padx=15, pady=5)
        tk.Label(icon_frame3, text="■", font=("Arial", 20), fg="white", bg="#f39c12").grid(row=0, column=0)

        self.avg_performance_label = tk.Label(card3, text="0%", font=("Arial", 24, "bold"),
                                              fg="white", bg="#f39c12")
        self.avg_performance_label.grid(row=2, column=0, sticky="w", padx=15, pady=(0, 15))
```

```
def create_top_students_section(self):
    """Create top 3 students section using grid layout"""
    # Section title
    title_label = tk.Label(self.dashboard_content, text="■ Top 3 Students (Les 3 Meilleurs Élèves)",
                           font=("Arial", 16, "bold"), fg="#2c3e50", bg="#f8f9fa")
    title_label.grid(row=3, column=0, columnspan=3, sticky="w", padx=20, pady=(20, 10))

    # Get top students data
    top_students = db.get_top_students(3)

    # Medal colors and positions
    medal_colors = ["#FFD700", "#C0C0C0", "#CD7F32"]  # Gold, Silver, Bronze
    medal_emojis = ["■", "■", "■"]
    positions = ["1st", "2nd", "3rd"]

    for i, student in enumerate(top_students):
        # Create student card
        card = tk.Frame(self.dashboard_content, bg="white", relief="solid", bd=1)
        card.grid(row=4, column=i, padx=10, pady=5, sticky="nsew")
        card.grid_columnconfigure(0, weight=1)
        card.grid_rowconfigure(0, weight=1)

        # Medal and position
        medal_frame = tk.Frame(card, bg="white")
        medal_frame.grid(row=0, column=0, sticky="ew", padx=15, pady=(15, 5))

        medal_label = tk.Label(medal_frame, text=medal_emojis[i], font=("Arial", 24),
                               bg="white", fg=medal_colors[i])
        medal_label.grid(row=0, column=0)

        position_label = tk.Label(medal_frame, text=positions[i], font=("Arial", 12, "bold"),
                                  bg="white", fg="#7f8c8d")
        position_label.grid(row=0, column=1, padx=(10, 0))

        # Student name
        name_label = tk.Label(card, text=student['fullname'], font=("Arial", 14, "bold"),
                              bg="white", fg="#2c3e50")
        name_label.grid(row=1, column=0, pady=(0, 5), sticky="ew")

        # Student details
        details_frame = tk.Frame(card, bg="white")
        details_frame.grid(row=2, column=0, sticky="ew", padx=15, pady=5)
        details_frame.grid_columnconfigure(0, weight=1)
        details_frame.grid_columnconfigure(1, weight=1)

        # CGPA
        cgpa_frame = tk.Frame(details_frame, bg="white")
        cgpa_frame.grid(row=0, column=0, columnspan=2, sticky="ew", pady=2)
        cgpa_frame.grid_columnconfigure(0, weight=1)
        cgpa_frame.grid_columnconfigure(1, weight=1)

        tk.Label(cgpa_frame, text="CGPA:", font=("Arial", 10), bg="white", fg="#7f8c8d").grid(row=0, colu
        cgpa_value = tk.Label(cgpa_frame, text=f"{student['cgpa']:.2f}", font=("Arial", 12, "bold"),
                              bg="white", fg="#27ae60")
        cgpa_value.grid(row=0, column=1, sticky="e")

        # Average percentage
        avg_frame = tk.Frame(details_frame, bg="white")
        avg_frame.grid(row=1, column=0, columnspan=2, sticky="ew", pady=2)
        avg_frame.grid_columnconfigure(0, weight=1)
        avg_frame.grid_columnconfigure(1, weight=1)

        tk.Label(avg_frame, text="Average:", font=("Arial", 10), bg="white", fg="#7f8c8d").grid(row=0, c
        avg_value = tk.Label(avg_frame, text=f"{student['avg_percentage']:.1f}%", font=("Arial", 12, "bo
                             bg="white", fg="#3498db")
        avg_value.grid(row=0, column=1, sticky="e")

        # Total exams
        exams_frame = tk.Frame(details_frame, bg="white")
        exams_frame.grid(row=2, column=0, columnspan=2, sticky="ew", pady=2)
        exams_frame.grid_columnconfigure(0, weight=1)
        exams_frame.grid_columnconfigure(1, weight=1)

        tk.Label(exams_frame, text="Exams:", font=("Arial", 10), bg="white", fg="#7f8c8d").grid(row=0, c
        exams_value = tk.Label(exams_frame, text=str(student['total_exams']), font=("Arial", 12, "bold")
                               bg="white", fg="#e74c3c")
        exams_value.grid(row=0, column=1, sticky="e")

        # Gender indicator
        gender_frame = tk.Frame(details_frame, bg="white")
        gender_frame.grid(row=3, column=0, columnspan=2, sticky="ew", pady=(5, 10))

        gender_emoji = "■" if student['gender'] == 'Male' else "■"
        tk.Label(gender_frame, text=gender_emoji, font=("Arial", 12), bg="white").grid(row=0, column=0,
        tk.Label(gender_frame, text=student['gender'], font=("Arial", 10), bg="white", fg="#7f8c8d").gri
    # If no students found, show message
```

```
        if not top_students:
            no_data_frame = tk.Frame(self.dashboard_content, bg="white", relief="solid", bd=1)
            no_data_frame.grid(row=4, column=0, columnspan=3, padx=10, pady=5, sticky="ew")

            tk.Label(no_data_frame, text="No student performance data available",
                    font=("Arial", 14), fg="#7f8c8d", bg="white").grid(row=0, column=0, pady=20, sticky="nsew

    def create_charts_section(self):
        """Create charts section with 2 columns layout using grid"""
        # Bar chart frame - Left column (Performance by Subject)
        bar_frame = tk.Frame(self.dashboard_content, bg="white", relief="solid", bd=1)
        bar_frame.grid(row=2, column=0, padx=10, pady=10, sticky="nsew")
        bar_frame.grid_columnconfigure(0, weight=1)
        bar_frame.grid_rowconfigure(1, weight=1)

        tk.Label(bar_frame, text="Performance by Subject", font=("Arial", 14, "bold"),
                fg="#2c3e50", bg="white").grid(row=0, column=0, pady=10, sticky="ew")

        # Enhanced bar chart using canvas
        self.bar_canvas = tk.Canvas(bar_frame, bg="white", highlightthickness=0)
        self.bar_canvas.grid(row=1, column=0, padx=20, pady=20, sticky="nsew")

        # Pie chart frame - Right column (Gender Distribution)
        pie_frame = tk.Frame(self.dashboard_content, bg="white", relief="solid", bd=1)
        pie_frame.grid(row=2, column=1, padx=10, pady=10, sticky="nsew")
        pie_frame.grid_columnconfigure(0, weight=1)
        pie_frame.grid_rowconfigure(1, weight=1)

        tk.Label(pie_frame, text="Gender Distribution", font=("Arial", 14, "bold"),
                fg="#2c3e50", bg="white").grid(row=0, column=0, pady=10, sticky="ew")

        # Enhanced pie chart using canvas
        self.pie_canvas = tk.Canvas(pie_frame, bg="white", highlightthickness=0)
        self.pie_canvas.grid(row=1, column=0, padx=20, pady=20, sticky="nsew")

        # Performance Trends chart - Bottom Left (spanning both columns)
        trends_frame = tk.Frame(self.dashboard_content, bg="white", relief="solid", bd=1)
        trends_frame.grid(row=2, column=2, padx=10, pady=10, sticky="nsew")
        trends_frame.grid_columnconfigure(0, weight=1)
        trends_frame.grid_rowconfigure(1, weight=1)

        tk.Label(trends_frame, text="Performance Trends", font=("Arial", 14, "bold"),
                fg="#2c3e50", bg="white").grid(row=0, column=0, pady=10, sticky="ew")

        # Performance trends chart
        self.trends_canvas = tk.Canvas(trends_frame, bg="white", highlightthickness=0)
        self.trends_canvas.grid(row=1, column=0, padx=20, pady=20, sticky="nsew")

    def create_students_content(self):
        """Create students management content using grid layout"""
        self.students_content = tk.Frame(self.content_frame, bg="#f8f9fa")

        # Configure grid weights for the students content
        self.students_content.grid_columnconfigure(0, weight=1)
        self.students_content.grid_rowconfigure(0, weight=0)  # Title row
        self.students_content.grid_rowconfigure(1, weight=0)  # Actions row
        self.students_content.grid_rowconfigure(2, weight=1)  # Table row

        # Title
        title_label = tk.Label(self.students_content, text="Student Management",
                               font=("Arial", 24, "bold"), fg="#2c3e50", bg="#f8f9fa")
        title_label.grid(row=0, column=0, sticky="w", padx=20, pady=(20, 10))

        # Search and actions frame
        actions_frame = tk.Frame(self.students_content, bg="#f8f9fa")
        actions_frame.grid(row=1, column=0, sticky="ew", padx=20, pady=10)
        actions_frame.grid_columnconfigure(1, weight=1)

        # Search
        tk.Label(actions_frame, text="Search:", font=("Arial", 11),
                fg="#2c3e50", bg="#f8f9fa").grid(row=0, column=0, sticky="w", padx=(0, 10))

        self.student_search_var = tk.StringVar()
        search_entry = tk.Entry(actions_frame, textvariable=self.student_search_var,
                                font=("Arial", 11), width=30)
        search_entry.grid(row=0, column=1, sticky="w", padx=(0, 20))
        search_entry.bind('<KeyRelease>', self.filter_students)

        # Action buttons frame
        buttons_frame = tk.Frame(actions_frame, bg="#f8f9fa")
        buttons_frame.grid(row=0, column=2, sticky="e")

        # Action buttons
        tk.Button(buttons_frame, text="■ Add Student", font=("Arial", 11),
                 fg="white", bg="#27ae60", relief="flat", cursor="hand2",
                 command=self.add_student).grid(row=0, column=0, padx=5, sticky="ew")
```

```
        tk.Button(buttons_frame, text="✏ Edit", font=("Arial", 11),
                  fg="white", bg="#3498db", relief="flat", cursor="hand2",
                  command=self.edit_student).grid(row=0, column=1, padx=5, sticky="ew")

        tk.Button(buttons_frame, text="🗑 Delete", font=("Arial", 11),
                  fg="white", bg="#e74c3c", relief="flat", cursor="hand2",
                  command=self.delete_student).grid(row=0, column=2, padx=5, sticky="ew")

        tk.Button(buttons_frame, text="📊 View Marks", font=("Arial", 11),
                  fg="white", bg="#9b59b6", relief="flat", cursor="hand2",
                  command=self.view_student_marks).grid(row=0, column=3, padx=5, sticky="ew")

        # Students table
        table_frame = tk.Frame(self.students_content, bg="white", relief="solid", bd=1)
        table_frame.grid(row=2, column=0, sticky="nsew", padx=20, pady=10)
        table_frame.grid_columnconfigure(0, weight=1)
        table_frame.grid_rowconfigure(0, weight=1)

        # Create Treeview
        columns = ('ID', 'Name', 'Email', 'Phone', 'Gender', 'Status', 'Enrollment Date')
        self.students_tree = ttk.Treeview(table_frame, columns=columns, show='headings', height=15)

        # Define headings
        for col in columns:
            self.students_tree.heading(col, text=col)
            self.students_tree.column(col, width=120)

        # Add scrollbar
        scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=self.students_tree.yview)
        self.students_tree.configure(yscrollcommand=scrollbar.set)

        self.students_tree.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
        scrollbar.grid(row=0, column=1, sticky="ns", pady=10)

    def create_teachers_content(self):
        """Create teachers management content using grid layout"""
        self.teachers_content = tk.Frame(self.content_frame, bg="#f8f9fa")

        # Configure grid weights for the teachers content
        self.teachers_content.grid_columnconfigure(0, weight=1)
        self.teachers_content.grid_rowconfigure(0, weight=0)  # Title row
        self.teachers_content.grid_rowconfigure(1, weight=0)  # Actions row
        self.teachers_content.grid_rowconfigure(2, weight=1)  # Table row

        # Title
        title_label = tk.Label(self.teachers_content, text="Teacher Management",
                               font=("Arial", 24, "bold"), fg="#2c3e50", bg="#f8f9fa")
        title_label.grid(row=0, column=0, sticky="w", padx=20, pady=(20, 10))

        # Search and actions frame
        actions_frame = tk.Frame(self.teachers_content, bg="#f8f9fa")
        actions_frame.grid(row=1, column=0, sticky="ew", padx=20, pady=10)
        actions_frame.grid_columnconfigure(1, weight=1)

        # Search
        tk.Label(actions_frame, text="Search:", font=("Arial", 11),
                 fg="#2c3e50", bg="#f8f9fa").grid(row=0, column=0, sticky="w", padx=(0, 10))

        self.teacher_search_var = tk.StringVar()
        search_entry = tk.Entry(actions_frame, textvariable=self.teacher_search_var,
                                font=("Arial", 11), width=30)
        search_entry.grid(row=0, column=1, sticky="w", padx=(0, 20))
        search_entry.bind('<KeyRelease>', self.filter_teachers)

        # Action buttons frame
        buttons_frame = tk.Frame(actions_frame, bg="#f8f9fa")
        buttons_frame.grid(row=0, column=2, sticky="e")

        # Action buttons
        tk.Button(buttons_frame, text="➕ Add Teacher", font=("Arial", 11),
                  fg="white", bg="#27ae60", relief="flat", cursor="hand2",
                  command=self.add_teacher).grid(row=0, column=0, padx=5, sticky="ew")

        tk.Button(buttons_frame, text="✏ Edit", font=("Arial", 11),
                  fg="white", bg="#3498db", relief="flat", cursor="hand2",
                  command=self.edit_teacher).grid(row=0, column=1, padx=5, sticky="ew")

        tk.Button(buttons_frame, text="🗑 Delete", font=("Arial", 11),
                  fg="white", bg="#e74c3c", relief="flat", cursor="hand2",
                  command=self.delete_teacher).grid(row=0, column=2, padx=5, sticky="ew")

        # Teachers table
        table_frame = tk.Frame(self.teachers_content, bg="white", relief="solid", bd=1)
        table_frame.grid(row=2, column=0, sticky="nsew", padx=20, pady=10)
        table_frame.grid_columnconfigure(0, weight=1)
        table_frame.grid_rowconfigure(0, weight=1)
```

```
        # Create Treeview
        columns = ('ID', 'Name', 'Email', 'Phone', 'Department', 'Qualification', 'Status')
        self.teachers_tree = ttk.Treeview(table_frame, columns=columns, show='headings', height=15)

        # Define headings
        for col in columns:
            self.teachers_tree.heading(col, text=col)
            self.teachers_tree.column(col, width=120)

        # Add scrollbar
        scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=self.teachers_tree.yview)
        self.teachers_tree.configure(yscrollcommand=scrollbar.set)

        self.teachers_tree.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
        scrollbar.grid(row=0, column=1, sticky="ns", pady=10)

    def create_settings_content(self):
        """Create settings content using grid layout"""
        self.settings_content = tk.Frame(self.content_frame, bg="#f8f9fa")

        # Configure grid weights for the settings content
        self.settings_content.grid_columnconfigure(0, weight=1)
        self.settings_content.grid_rowconfigure(0, weight=0)  # Title row
        self.settings_content.grid_rowconfigure(1, weight=1)  # Content row

        # Title
        title_label = tk.Label(self.settings_content, text="System Settings",
                               font=("Arial", 24, "bold"), fg="#2c3e50", bg="#f8f9fa")
        title_label.grid(row=0, column=0, sticky="w", padx=20, pady=(20, 10))

        # Settings content
        settings_frame = tk.Frame(self.settings_content, bg="white", relief="solid", bd=1)
        settings_frame.grid(row=1, column=0, sticky="nsew", padx=20, pady=10)
        settings_frame.grid_columnconfigure(0, weight=1)
        settings_frame.grid_rowconfigure(0, weight=1)

        tk.Label(settings_frame, text="Settings functionality will be implemented here",
                font=("Arial", 14), fg="#7f8c8d", bg="white").grid(row=0, column=0, sticky="nsew")

    def highlight_menu(self, active_menu):
        """Highlight active menu item"""
        for text, btn in self.menu_buttons.items():
            if text == active_menu:
                btn.config(bg="#ecf0f1", fg="#2c3e50")
            else:
                btn.config(bg="white", fg="#2c3e50")

    def show_dashboard(self):
        """Show dashboard content"""
        self.hide_all_content()
        self.dashboard_content.grid(row=0, column=0, sticky="nsew")
        self.highlight_menu("■ Dashboard")

    def show_students(self):
        """Show students content"""
        self.hide_all_content()
        self.students_content.grid(row=0, column=0, sticky="nsew")
        self.highlight_menu("■ Manage Students")

    def show_teachers(self):
        """Show teachers content"""
        self.hide_all_content()
        self.teachers_content.grid(row=0, column=0, sticky="nsew")
        self.highlight_menu("■■■ Manage Teachers")

    def show_predictions(self):
        """Show predictions content"""
        self.hide_all_content()
        self.predictions_content.grid(row=0, column=0, sticky="nsew")
        self.highlight_menu("■ Predictions")

    def show_settings(self):
        """Show settings content"""
        self.hide_all_content()
        self.settings_content.grid(row=0, column=0, sticky="nsew")
        self.highlight_menu("■■ Settings")

    def hide_all_content(self):
        """Hide all content frames"""
        self.dashboard_content.grid_forget()
        self.students_content.grid_forget()
        self.teachers_content.grid_forget()
        if hasattr(self, 'predictions_content'):
            self.predictions_content.grid_forget()
        self.settings_content.grid_forget()
    def create_predictions_content(self):
```

```
        """Create predictions management page"""
        self.predictions_content = tk.Frame(self.content_frame, bg="#f8f9fa")
        self.predictions_content.grid_columnconfigure(0, weight=1)
        self.predictions_content.grid_rowconfigure(0, weight=0)
        self.predictions_content.grid_rowconfigure(1, weight=0)
        self.predictions_content.grid_rowconfigure(2, weight=1)  # students list
        self.predictions_content.grid_rowconfigure(3, weight=1)  # results box

        title_label = tk.Label(self.predictions_content, text="AI Predictions",
                               font=("Arial", 24, "bold"), fg="#2c3e50", bg="#f8f9fa")
        title_label.grid(row=0, column=0, sticky="w", padx=20, pady=(20, 10))

        actions = tk.Frame(self.predictions_content, bg="#f8f9fa")
        actions.grid(row=1, column=0, sticky="ew", padx=20, pady=10)
        actions.grid_columnconfigure(3, weight=1)

        tk.Button(actions, text="■ Predict Selected Student", font=("Arial", 11), fg="white", bg="#3498db",
                  relief="flat", cursor="hand2", command=self.predict_selected_student).grid(row=0, column=0
        tk.Button(actions, text="■ Clear", font=("Arial", 11), fg="#2c3e50", bg="#ecf0f1",
                  relief="flat", cursor="hand2", command=self.clear_predictions).grid(row=0, column=1, padx=

        # Students selector panel
        sel = tk.Frame(self.predictions_content, bg="#f8f9fa")
        sel.grid(row=2, column=0, sticky="nsew", padx=20, pady=(0, 10))
        sel.grid_columnconfigure(0, weight=1)
        sel.grid_rowconfigure(1, weight=1)

        tk.Label(sel, text="Select Student", font=("Arial", 14, "bold"), fg="#2c3e50", bg="#f8f9fa").grid(ro

        table_frame = tk.Frame(sel, bg="white", relief="solid", bd=1)
        table_frame.grid(row=1, column=0, sticky="nsew")
        table_frame.grid_columnconfigure(0, weight=1)
        table_frame.grid_rowconfigure(0, weight=1)

        cols = ("ID", "Name", "Email")
        self.pred_students_tree = ttk.Treeview(table_frame, columns=cols, show='headings', height=8)
        for c in cols:
            self.pred_students_tree.heading(c, text=c)
            self.pred_students_tree.column(c, width=160)
        vs = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=self.pred_students_tree.yview)
        self.pred_students_tree.configure(yscrollcommand=vs.set)
        self.pred_students_tree.grid(row=0, column=0, sticky="nsew")
        vs.grid(row=0, column=1, sticky="ns")

        # Load students into selector
        try:
            for item in self.pred_students_tree.get_children():
                self.pred_students_tree.delete(item)
            _students = db.get_all_students() or []
            for s in _students:
                self.pred_students_tree.insert('', 'end', values=(s.get('student_id'), s.get('fullname'), s.
        except Exception:
            pass

        # Auto-predict on selection
        try:
            self.pred_students_tree.bind('<<TreeviewSelect>>', lambda e: self.predict_selected_student())
        except Exception:
            pass

        # Results and chart area
        # Chart container (matplotlib embedded if available)
        chart_wrap = tk.Frame(self.predictions_content, bg="#f8f9fa")
        chart_wrap.grid(row=3, column=0, sticky="nsew", padx=20, pady=10)
        chart_wrap.grid_columnconfigure(0, weight=1)
        chart_wrap.grid_columnconfigure(1, weight=1)
        chart_wrap.grid_rowconfigure(0, weight=1)

        # Left: bar chart (per-subject predicted %)
        self.pred_bar_frame = tk.Frame(chart_wrap, bg="white", relief="solid", bd=1)
        self.pred_bar_frame.grid(row=0, column=0, sticky="nsew", padx=(0, 5))
        self.pred_bar_frame.grid_columnconfigure(0, weight=1)
        self.pred_bar_frame.grid_rowconfigure(0, weight=1)

        # Right: pie chart (grade distribution)
        self.pred_pie_frame = tk.Frame(chart_wrap, bg="white", relief="solid", bd=1)
        self.pred_pie_frame.grid(row=0, column=1, sticky="nsew", padx=(5, 0))
        self.pred_pie_frame.grid_columnconfigure(0, weight=1)
        self.pred_pie_frame.grid_rowconfigure(0, weight=1)

        # Text results below chart
        self.pred_results = tk.Text(self.predictions_content, height=8, bg="white")
        self.pred_results.grid(row=4, column=0, sticky="nsew", padx=20, pady=(0, 10))

    def clear_predictions(self):
        try:
```

```
            self.pred_results.delete('1.0', 'end')
        except Exception:
            pass
        # Clear chart areas
        try:
            for w in getattr(self, 'pred_bar_frame', []).winfo_children():
                w.destroy()
        except Exception:
            pass
        try:
            for w in getattr(self, 'pred_pie_frame', []).winfo_children():
                w.destroy()
        except Exception:
            pass

    def predict_selected_student(self):
        student_id = None
        # Prefer selection from Predictions page student list
        if hasattr(self, 'pred_students_tree'):
            sel = self.pred_students_tree.selection()
            if sel:
                try:
                    student_id = self.pred_students_tree.item(sel[0])['values'][0]
                except Exception:
                    student_id = None
        # Fallback to Students tab selection
        if student_id is None:
            selection = self.students_tree.selection() if hasattr(self, 'students_tree') else []
            if selection:
                item = self.students_tree.item(selection[0])
                student_id = item['values'][0]
        if student_id is None:
            messagebox.showwarning("Warning", "Please select a student (in Predictions or Students tab).")
            return
        try:
            from ml_model import load_model, predict_next_percentage, percentage_to_grade
            model = load_model()
            if not model:
                messagebox.showwarning("Warning", "Model not available. Please train it once from terminal."
                return
            # Predict for each subject the student has marks in
            marks = db.get_student_marks(student_id) or []
            if not marks:
                messagebox.showinfo("Info", "No marks found for this student.")
                self.clear_predictions()
                return
            # Build subject list (id -> name)
            subj_seen = {}
            for m in marks:
                sid = m.get('subject_id')
                name = m.get('subject_name')
                if sid is not None and name:
                    subj_seen[int(sid)] = str(name)
            preds = []
            for sid, name in subj_seen.items():
                try:
                    p = predict_next_percentage(model, int(student_id), int(sid))
                    if p is not None:
                        preds.append((name, float(p)))
                except Exception:
                    continue
            preds.sort(key=lambda x: x[0])
            # Update text results
            self.pred_results.delete('1.0', 'end')
            if not preds:
                self.pred_results.insert('end', "Not enough data to predict for this student.\n")
            else:
                self.pred_results.insert('end', f"Predictions for student {student_id}:\n")
                for name, p in preds:
                    self.pred_results.insert('end', f" - {name}: {p:.1f}% ({percentage_to_grade(p)})\n")
                self.pred_results.see('end')
            # Render chart
            self.render_pred_charts(preds)
        except Exception as e:
            messagebox.showerror("Error", f"Prediction failed: {e}")

    def render_pred_charts(self, preds):
        # Clear previous charts
        try:
            for w in self.pred_bar_frame.winfo_children():
                w.destroy()
        except Exception:
            pass
        try:
            for w in self.pred_pie_frame.winfo_children():
                w.destroy()
```

```
        except Exception:
            pass
        # Empty state
        if not preds:
            for frame in (self.pred_bar_frame, self.pred_pie_frame):
                try:
                    tk.Label(frame, text="No predictions to display", bg="white", fg="#7f8c8d", font=("Arial
                except Exception:
                    pass
            return
        # Try to render charts
        try:
            import matplotlib.pyplot as plt
            from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
        except ImportError:
            for frame in (self.pred_bar_frame, self.pred_pie_frame):
                tk.Label(frame, text="Matplotlib not available", bg="white", fg="#7f8c8d", font=("Arial", 12
            return

        # Bar chart (left)
        try:
            fig_bar, ax = plt.subplots(figsize=(6.5, 3.2))
            fig_bar.patch.set_facecolor('white')
            names = [n for n, _ in preds]
            values = [v for _, v in preds]
            xs = range(len(names))
            bars = ax.bar(xs, values, color='#3498db')
            ax.set_ylim(0, 100)
            ax.set_ylabel('Predicted %')
            ax.set_title('Predicted Next Performance by Subject')
            ax.set_xticks(list(xs))
            ax.set_xticklabels(names, rotation=20, ha='right')
            for b, v in zip(bars, values):
                ax.text(b.get_x() + b.get_width()/2, b.get_height()+1, f"{v:.1f}%", ha='center', va='bottom'
            plt.tight_layout()
            canvas_bar = FigureCanvasTkAgg(fig_bar, master=self.pred_bar_frame)
            canvas_bar.draw()
            canvas_bar.get_tk_widget().grid(row=0, column=0, sticky="nsew")
        except Exception:
            tk.Label(self.pred_bar_frame, text="Failed to render bar chart", bg="white", fg="#e74c3c", font=

        # Pie chart (right): grade distribution
        try:
            from ml_model import percentage_to_grade
            grades = [percentage_to_grade(v) for _, v in preds]
            order = ['A+', 'A', 'B', 'C', 'D', 'F']
            counts = [grades.count(g) for g in order]
            if sum(counts) == 0:
                tk.Label(self.pred_pie_frame, text="No data for grade distribution", bg="white", fg="#7f8c8d
            else:
                fig_pie, axp = plt.subplots(figsize=(6.5, 3.2))
                fig_pie.patch.set_facecolor('white')
                colors = ['#2ecc71', '#27ae60', '#3498db', '#f1c40f', '#e67e22', '#e74c3c']
                wedges, texts, autotexts = axp.pie(counts, labels=order, autopct='%1.0f%%', startangle=90, c
                centre_circle = plt.Circle((0,0), 0.55, fc='white')
                axp.add_artist(centre_circle)
                axp.set_title('Predicted Grade Distribution')
                plt.tight_layout()
                canvas_pie = FigureCanvasTkAgg(fig_pie, master=self.pred_pie_frame)
                canvas_pie.draw()
                canvas_pie.get_tk_widget().grid(row=0, column=0, sticky="nsew")
        except Exception:
            tk.Label(self.pred_pie_frame, text="Failed to render pie chart", bg="white", fg="#e74c3c", font=

    def predict_student_subject_dialog(self):
        # Deprecated in simplified UI; keep no-op for compatibility
        messagebox.showinfo("Info", "Use the student list to select and predict.")

    def load_dashboard_data(self):
        """Load dashboard statistics"""
        stats = db.get_system_stats()
        # Update visible labels (no undefined stats_cards)
        self.students_count_label.config(text=str(stats.get('active_students', 0)))
        self.teachers_count_label.config(text=str(stats.get('active_teachers', 0)))
        self.avg_performance_label.config(text=f"{stats.get('average_marks', 0)}%")
        # Refresh all charts
        self.create_bar_chart()
        self.create_pie_chart()
        self.create_trends_chart()
        # Refresh top students section
        self.refresh_top_students()

    def create_bar_chart(self):
        """Create bar chart using subject average percentages from DB"""
        self.bar_canvas.delete("all")
        # Get canvas dimensions
```

```
        canvas_width = self.bar_canvas.winfo_width()
        canvas_height = self.bar_canvas.winfo_height()

        if canvas_width <= 1 or canvas_height <= 1:
            # Canvas not yet rendered, schedule for later
            self.root.after(100, self.create_bar_chart)
            return

        # Real data: subject average percentages
        rows = db.get_subject_average_percentages(limit=8)
        subjects = [r['subject_name'] for r in rows] or ["No Data"]
        values = [round(r['avg_pct'] or 0, 1) for r in rows] or [0]
        colors = ["#3498db", "#e74c3c", "#f39c12", "#27ae60", "#9b59b6", "#16a085", "#d35400", "#2ecc71"]

        # Calculate dimensions
        margin = 40
        chart_width = canvas_width - 2 * margin
        chart_height = canvas_height - 2 * margin
        bar_width = (chart_width - (len(subjects) - 1) * 20) // len(subjects)

        # Draw bars
        for i, (subject, value, color) in enumerate(zip(subjects, values, colors)):
            x = margin + i * (bar_width + 20)
            bar_height = (value / 100) * chart_height
            y = margin + chart_height - bar_height

            # Bar with gradient effect
            self.bar_canvas.create_rectangle(x, y, x + bar_width, margin + chart_height,
                                             fill=color, outline="", width=0)

            # Value label on top
            self.bar_canvas.create_text(x + bar_width/2, y - 15,
                                        text=f"{value}%", font=("Arial", 11, "bold"),
                                        fill="#2c3e50")

            # Subject label at bottom
            self.bar_canvas.create_text(x + bar_width/2, margin + chart_height + 20,
                                        text=subject, font=("Arial", 10, "bold"),
                                        fill="#2c3e50")

        # Add grid lines
        for i in range(0, 101, 20):
            y = margin + chart_height - (i / 100) * chart_height
            self.bar_canvas.create_line(margin, y, margin + chart_width, y,
                                        fill="#ecf0f1", width=1)

    def create_pie_chart(self):
        """Create gender distribution donut chart from DB"""
        self.pie_canvas.delete("all")

        # Get canvas dimensions
        canvas_width = self.pie_canvas.winfo_width()
        canvas_height = self.pie_canvas.winfo_height()

        if canvas_width <= 1 or canvas_height <= 1:
            # Canvas not yet rendered, schedule for later
            self.root.after(100, self.create_pie_chart)
            return

        # Real gender distribution
        gd = db.get_gender_distribution()
        male_count = gd.get('Male', 0)
        female_count = gd.get('Female', 0)
        total = male_count + female_count

        center_x = canvas_width // 2
        center_y = canvas_height // 2
        radius = min(canvas_width, canvas_height) // 4

        # Draw pie chart
        if total > 0:
            male_angle = (male_count / total) * 360
            female_angle = (female_count / total) * 360

            # Male slice
            self.pie_canvas.create_arc(center_x - radius, center_y - radius,
                                       center_x + radius, center_y + radius,
                                       start=0, extent=male_angle, fill="#3498db", outline="white", width=2)

            # Female slice
            self.pie_canvas.create_arc(center_x - radius, center_y - radius,
                                       center_x + radius, center_y + radius,
                                       start=male_angle, extent=female_angle, fill="#e74c3c", outline="white",

            # Center circle for donut effect
            self.pie_canvas.create_oval(center_x - radius//2, center_y - radius//2,
```

```
                                    center_x + radius//2, center_y + radius//2,
                                    fill="white", outline="")

        # Enhanced legend
        legend_x = 20
        legend_y = 20

        # Male legend
        self.pie_canvas.create_rectangle(legend_x, legend_y, legend_x + 15, legend_y + 15,
                                         fill="#3498db", outline="")
        self.pie_canvas.create_text(legend_x + 25, legend_y + 8, text=f"Male: {male_count}",
                                    font=("Arial", 10, "bold"), fill="#2c3e50")

        # Female legend
        self.pie_canvas.create_rectangle(legend_x, legend_y + 25, legend_x + 15, legend_y + 40,
                                         fill="#e74c3c", outline="")
        self.pie_canvas.create_text(legend_x + 25, legend_y + 33, text=f"Female: {female_count}",
                                    font=("Arial", 10, "bold"), fill="#2c3e50")

        # Total in center
        self.pie_canvas.create_text(center_x, center_y, text=f"Total\n{total}",
                                    font=("Arial", 12, "bold"), fill="#2c3e50", justify="center")

    def create_trends_chart(self):
        """Create performance trends line chart from DB monthly averages"""
        self.trends_canvas.delete("all")

        # Get canvas dimensions
        canvas_width = self.trends_canvas.winfo_width()
        canvas_height = self.trends_canvas.winfo_height()

        if canvas_width <= 1 or canvas_height <= 1:
            # Canvas not yet rendered, schedule for later
            self.root.after(100, self.create_trends_chart)
            return

        # Real monthly averages (last 6 months)
        rows = db.get_monthly_trends_average(months=6)
        months = [r['ym'] for r in rows] or ["N/A"]
        values = [round(r['avg_pct'] or 0, 1) for r in rows] or [0]

        margin = 40
        chart_width = canvas_width - 2 * margin
        chart_height = canvas_height - 2 * margin

        # Draw grid lines
        for i in range(0, 101, 20):
            y = margin + chart_height - (i / 100) * chart_height
            self.trends_canvas.create_line(margin, y, margin + chart_width, y,
                                           fill="#ecf0f1", width=1)

        # Draw trend line
        points = []
        for i, (month, value) in enumerate(zip(months, values)):
            x = margin + (i / (len(months) - 1)) * chart_width
            y = margin + chart_height - (value / 100) * chart_height
            points.extend([x, y])

            # Draw data points
            self.trends_canvas.create_oval(x - 4, y - 4, x + 4, y + 4,
                                           fill="#3498db", outline="white", width=2)

            # Month labels
            self.trends_canvas.create_text(x, margin + chart_height + 20,
                                           text=month, font=("Arial", 9), fill="#2c3e50")

        # Draw trend line
        if len(points) >= 4:
            self.trends_canvas.create_line(points, fill="#3498db", width=3, smooth=True)

        # Y-axis labels
        for i in range(0, 101, 20):
            y = margin + chart_height - (i / 100) * chart_height
            self.trends_canvas.create_text(margin - 10, y, text=f"{i}%",
                                           font=("Arial", 9), fill="#7f8c8d", anchor="e")


    def refresh_top_students(self):
        """Refresh the top students section with updated data"""
        # Find and destroy existing top students widgets
        for widget in self.dashboard_content.winfo_children():
            if isinstance(widget, tk.Label) and "Top 3 Students" in widget.cget("text"):
                # Destroy the title and all student cards
                widget.destroy()
                # Find and destroy all student cards in row 4
                for child in self.dashboard_content.winfo_children():
```

```
                if isinstance(child, tk.Frame) and child.grid_info().get('row') == 4:
                    child.destroy()
            break

    # Recreate the top students section
    self.create_top_students_section()

def load_students(self):
    """Load students data"""
    # Clear existing items
    for item in self.students_tree.get_children():
        self.students_tree.delete(item)

    students = db.get_all_students()
    for student in students:
        self.students_tree.insert('', 'end', values=(
            student['student_id'],
            student['fullname'],
            student['email'],
            student['phone'],
            student['gender'],
            student['status']
        ))

def load_teachers(self):
    """Load teachers data"""
    # Clear existing items
    for item in self.teachers_tree.get_children():
        self.teachers_tree.delete(item)

    teachers = db.get_all_teachers()
    for teacher in teachers:
        self.teachers_tree.insert('', 'end', values=(
            teacher['teacher_id'],
            teacher['fullname'],
            teacher['email'],
            teacher['phone'],
            teacher['department'],
            teacher['status']
        ))

def filter_students(self, *args):
    """Filter students based on search"""
    search_term = self.student_search_var.get().lower()

    # Clear existing items
    for item in self.students_tree.get_children():
        self.students_tree.delete(item)

    students = db.get_all_students()
    for student in students:
        if (search_term in student['fullname'].lower() or
            search_term in student['email'].lower() or
            search_term in str(student['student_id'])):
            self.students_tree.insert('', 'end', values=(
                student['student_id'],
                student['fullname'],
                student['email'],
                student['phone'],
                student['gender'],
                student['status']
            ))

def filter_teachers(self, *args):
    """Filter teachers based on search"""
    search_term = self.teacher_search_var.get().lower()

    # Clear existing items
    for item in self.teachers_tree.get_children():
        self.teachers_tree.delete(item)

    teachers = db.get_all_teachers()
    for teacher in teachers:
        if (search_term in teacher['fullname'].lower() or
            search_term in teacher['email'].lower() or
            search_term in str(teacher['teacher_id'])):
            self.teachers_tree.insert('', 'end', values=(
                teacher['teacher_id'],
                teacher['fullname'],
                teacher['email'],
                teacher['phone'],
                teacher['department'],
                teacher['status']
            ))

def add_student(self):
```

```
"""Add new student - persists to DB and refreshes table"""
form = tk.Toplevel(self.root)
form.title("Add New Student")
form.geometry("480x560")
form.resizable(False, False)

# Form variables
username_var = tk.StringVar()
password_var = tk.StringVar()
fullname_var = tk.StringVar()
email_var = tk.StringVar()
phone_var = tk.StringVar()
dob_var = tk.StringVar()
gender_var = tk.StringVar(value="Male")
address_var = tk.StringVar()
status_var = tk.StringVar(value="Active")

row = 0
def add_row(label_text, widget):
    tk.Label(form, text=label_text, font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sti
    widget.grid(row=row, column=1, padx=20, pady=8, sticky="w")
    return 1

username_entry = tk.Entry(form, textvariable=username_var, font=("Arial", 11))
password_entry = tk.Entry(form, textvariable=password_var, font=("Arial", 11), show='*')
fullname_entry = tk.Entry(form, textvariable=fullname_var, font=("Arial", 11))
email_entry = tk.Entry(form, textvariable=email_var, font=("Arial", 11))
phone_entry = tk.Entry(form, textvariable=phone_var, font=("Arial", 11))
dob_entry = tk.Entry(form, textvariable=dob_var, font=("Arial", 11))

row += add_row("Username", username_entry)
row += add_row("Password", password_entry)
row += add_row("Full Name", fullname_entry)
row += add_row("Email", email_entry)
row += add_row("Phone", phone_entry)
row += add_row("Date of Birth (YYYY-MM-DD)", dob_entry)

# Gender radios
tk.Label(form, text="Gender", font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sticky="w
gender_frame = tk.Frame(form)
gender_frame.grid(row=row, column=1, padx=20, pady=8, sticky="w")
tk.Radiobutton(gender_frame, text="Male", variable=gender_var, value="Male").pack(side="left")
tk.Radiobutton(gender_frame, text="Female", variable=gender_var, value="Female").pack(side="left")
row += 1

address_entry = tk.Entry(form, textvariable=address_var, font=("Arial", 11))
row += add_row("Address", address_entry)

# Status combobox
tk.Label(form, text="Status", font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sticky="w
status_combo = ttk.Combobox(form, textvariable=status_var, values=["Active", "Inactive"], state="rea
status_combo.grid(row=row, column=1, padx=20, pady=8, sticky="w")
row += 1

def submit():
    username = username_entry.get().strip()
    password = password_entry.get().strip()
    fullname = fullname_entry.get().strip()
    email = email_entry.get().strip()
    phone = phone_entry.get().strip()
    dob = dob_entry.get().strip()
    gender = gender_var.get().strip() or "Male"
    address = address_entry.get().strip()
    status = status_var.get().strip()

    # Core required fields only
    required_map = {
        "username": username,
        "password": password,
        "full name": fullname,
        "email": email,
    }
    missing = [field for field, value in required_map.items() if not value]
    if missing:
        messagebox.showerror("Error", f"Please fill: {', '.join(missing)}.", parent=form)
        return

    # Normalize optional fields
    phone = phone or None
    address = address or None

    # Normalize date of birth: accept DD-MM-YYYY and convert; allow empty -> NULL
    if dob:
        if len(dob) == 10 and dob[2] == '-' and dob[5] == '-':
            # DD-MM-YYYY -> YYYY-MM-DD
            try:
```

```
                    d, m, y = dob.split('-')
                    int(d); int(m); int(y)
                    dob = f"{y}-{m.zfill(2)}-{d.zfill(2)}"
                except Exception:
                    messagebox.showerror("Error", "Invalid date of birth. Use YYYY-MM-DD or DD-MM-YYYY."
                    return
            elif len(dob) == 10 and dob[4] == '-' and dob[7] == '-':
                # Looks like YYYY-MM-DD, accept as is
                pass
            else:
                messagebox.showerror("Error", "Invalid date of birth. Use YYYY-MM-DD or DD-MM-YYYY.", pa
                return
        else:
            dob = None

        # Duplicate checks aligned with DB constraints (users.username UNIQUE, students.email UNIQUE)
        try:
            existing_user = db.check_username_exists(username)
        except Exception:
            existing_user = db.execute_query("SELECT user_id FROM users WHERE username = %s", (username,
        if existing_user:
            messagebox.showerror("Error", "Username already exists. Please choose another username.", pa
            return

        if email:
            existing_email = db.execute_query("SELECT student_id FROM students WHERE email = %s", (email
            if existing_email:
                messagebox.showerror("Error", "Email already exists. Please use a different email.", par
                return

        # Persist to DB
        ok = db.add_student(username, password, fullname, email, phone, dob, gender, address, status)
        if ok:
            messagebox.showinfo("Success", "Student added successfully.", parent=form)
            # refresh table and dashboard
            self.load_students()
            self.load_dashboard_data()
            form.destroy()
        else:
            messagebox.showerror("Error", "Failed to add student. Username or email may already exist.",

    tk.Button(form, text="Add Student", font=("Arial", 12, "bold"), bg="#27ae60", fg="white", command=su

    form.transient(self.root)
    form.grab_set()
    self.root.wait_window(form)

def edit_student(self):
    """Edit selected student"""
    selection = self.students_tree.selection()
    if not selection:
        messagebox.showwarning("Warning", "Please select a student to edit")
        return
    item = self.students_tree.item(selection[0])
    values = item['values']
    form = tk.Toplevel(self.root)
    form.title("Edit Student")
    form.geometry("520x520")
    form.resizable(False, False)

    # Fetch additional fields not in the table
    student_id = values[0]
    current = db.execute_query(
        "SELECT fullname, email, phone, date_of_birth, gender, address, status FROM students WHERE stude
        (student_id,)
    )
    cur = current[0] if current else {}

    fullname_value = cur.get('fullname', values[1])
    email_value = cur.get('email', values[2])
    phone_value = cur.get('phone', values[3])
    # Format DOB to YYYY-MM-DD if it's a date object
    dob_value = cur.get('date_of_birth')
    if hasattr(dob_value, 'strftime'):
        dob_value = dob_value.strftime('%Y-%m-%d')
    dob_value = dob_value or ''
    gender_var = tk.StringVar(value=cur.get('gender', values[4]))
    address_value = cur.get('address', '') or ''
    status_var = tk.StringVar(value=cur.get('status', values[5]))

    row = 0
    def add_row(label_text, widget):
        tk.Label(form, text=label_text, font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sti
        widget.grid(row=row, column=1, padx=20, pady=8, sticky="w")
        return 1
    fullname_entry = tk.Entry(form, font=("Arial", 11))
```

```
        fullname_entry.insert(0, str(fullname_value or ""))
        row += add_row("Full Name", fullname_entry)
        email_entry = tk.Entry(form, font=("Arial", 11))
        email_entry.insert(0, str(email_value or ""))
        row += add_row("Email", email_entry)
        phone_entry = tk.Entry(form, font=("Arial", 11))
        phone_entry.insert(0, str(phone_value or ""))
        row += add_row("Phone", phone_entry)
        dob_entry = tk.Entry(form, font=("Arial", 11))
        dob_entry.insert(0, str(dob_value))
        row += add_row("Date of Birth (YYYY-MM-DD)", dob_entry)

        # Gender radios
        tk.Label(form, text="Gender", font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sticky="w
        gender_frame = tk.Frame(form)
        gender_frame.grid(row=row, column=1, padx=20, pady=8, sticky="w")
        tk.Radiobutton(gender_frame, text="Male", variable=gender_var, value="Male").pack(side="left")
        tk.Radiobutton(gender_frame, text="Female", variable=gender_var, value="Female").pack(side="left")
        tk.Radiobutton(gender_frame, text="Other", variable=gender_var, value="Other").pack(side="left")
        row += 1

        address_entry = tk.Entry(form, font=("Arial", 11))
        address_entry.insert(0, str(address_value))
        row += add_row("Address", address_entry)

        # Status combobox
        tk.Label(form, text="Status", font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sticky="w
        status_combo = ttk.Combobox(form, textvariable=status_var, values=["Active", "Inactive", "Graduated"
        status_combo.grid(row=row, column=1, padx=20, pady=8, sticky="w")
        row += 1

        def submit():
            full_name = fullname_entry.get().strip()
            email = email_entry.get().strip()
            phone = phone_entry.get().strip()
            dob = dob_entry.get().strip()
            gender = gender_var.get().strip() or "Male"
            address = address_entry.get().strip() or None
            status = status_var.get().strip()

            if not full_name or not email:
                messagebox.showerror("Error", "Full name and email are required.", parent=form)
                return

            # Normalize date input
            if dob:
                if len(dob) == 10 and dob[2] == '-' and dob[5] == '-':
                    try:
                        d, m, y = dob.split('-')
                        int(d); int(m); int(y)
                        dob = f"{y}-{m.zfill(2)}-{d.zfill(2)}"
                    except Exception:
                        messagebox.showerror("Error", "Invalid date of birth. Use YYYY-MM-DD or DD-MM-YYYY."
                        return
                elif len(dob) == 10 and dob[4] == '-' and dob[7] == '-':
                    pass
                else:
                    messagebox.showerror("Error", "Invalid date of birth. Use YYYY-MM-DD or DD-MM-YYYY.", pa
                    return
            else:
                dob = None

            if db.update_student(student_id, full_name, email, phone or None, dob, gender, address, status):
                messagebox.showinfo("Success", "Student updated successfully.", parent=form)
                self.load_students()
                form.destroy()
            else:
                messagebox.showerror("Error", "Failed to update student.", parent=form)

        tk.Button(form, text="Update Student", font=("Arial", 12, "bold"), bg="#3498db", fg="white", command

        form.transient(self.root)
        form.grab_set()
        self.root.wait_window(form)

    def delete_student(self):
        """Delete selected student"""
        selection = self.students_tree.selection()
        if not selection:
            messagebox.showwarning("Warning", "Please select a student to delete")
            return

        if messagebox.askyesno("Confirm", "Are you sure you want to delete this student?"):
            item = self.students_tree.item(selection[0])
            student_id = item['values'][0]
            if db.delete_student(student_id):
```

```
            messagebox.showinfo("Success", "Student deleted successfully")
            self.load_students()
            self.load_dashboard_data()
        else:
            messagebox.showerror("Error", "Failed to delete student")

def view_student_marks(self):
    """View marks for selected student in a modal table"""
    selection = self.students_tree.selection()
    if not selection:
        messagebox.showwarning("Warning", "Please select a student to view marks")
        return
    item = self.students_tree.item(selection[0])
    values = item['values']
    student_id = values[0]
    student_name = values[1]

    # Fetch marks
    marks = db.get_student_marks(student_id) or []

    dialog = tk.Toplevel(self.root)
    dialog.title(f"Marks - {student_name}")
    dialog.geometry("850x500")
    dialog.resizable(True, True)

    container = tk.Frame(dialog, bg="white")
    container.pack(fill="both", expand=True, padx=10, pady=10)

    cols = ("Exam Date", "Subject", "Subject Code", "Teacher", "Marks Obtained", "Total", "Percent")
    tree = ttk.Treeview(container, columns=cols, show='headings', height=16)
    for col in cols:
        tree.heading(col, text=col)
        tree.column(col, width=120)

    vs = ttk.Scrollbar(container, orient=tk.VERTICAL, command=tree.yview)
    tree.configure(yscrollcommand=vs.set)
    tree.grid(row=0, column=0, sticky="nsew")
    vs.grid(row=0, column=1, sticky="ns")
    container.grid_columnconfigure(0, weight=1)
    container.grid_rowconfigure(0, weight=1)

    # Populate
    for m in marks:
        exam_date = m.get('exam_date')
        if hasattr(exam_date, 'strftime'):
            exam_date = exam_date.strftime('%Y-%m-%d')
        percent = 0
        try:
            if m.get('total_marks'):
                percent = round((m.get('marks_obtained', 0) / m.get('total_marks')) * 100, 2)
        except Exception:
            percent = 0
        tree.insert('', 'end', values=(
            exam_date or '',
            m.get('subject_name', ''),
            m.get('subject_code', ''),
            m.get('teacher_name', ''),
            m.get('marks_obtained', ''),
            m.get('total_marks', ''),
            f"{percent}%"
        ))

    # Close button
    tk.Button(dialog, text="Close", command=dialog.destroy, bg="#e0e0e0").pack(pady=8)

    dialog.transient(self.root)
    dialog.grab_set()
    self.root.wait_window(dialog)

def add_teacher(self):
    """Add new teacher - persists to DB and refreshes table"""
    form = tk.Toplevel(self.root)
    form.title("Add New Teacher")
    form.geometry("520x520")
    form.resizable(False, False)

    # Form variables
    username_var = tk.StringVar()
    password_var = tk.StringVar()
    fullname_var = tk.StringVar()
    email_var = tk.StringVar()
    phone_var = tk.StringVar()
    department_var = tk.StringVar()
    qualification_var = tk.StringVar()
    status_var = tk.StringVar(value="Active")
    row = 0
```

```
        def add_row(label_text, widget):
            tk.Label(form, text=label_text, font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sti
            widget.grid(row=row, column=1, padx=20, pady=8, sticky="w")
            return 1

        # Keep Entry refs
        username_entry = tk.Entry(form, textvariable=username_var, font=("Arial", 11))
        password_entry = tk.Entry(form, textvariable=password_var, font=("Arial", 11), show='*')
        fullname_entry = tk.Entry(form, textvariable=fullname_var, font=("Arial", 11))
        email_entry = tk.Entry(form, textvariable=email_var, font=("Arial", 11))
        phone_entry = tk.Entry(form, textvariable=phone_var, font=("Arial", 11))
        department_entry = tk.Entry(form, textvariable=department_var, font=("Arial", 11))
        qualification_entry = tk.Entry(form, textvariable=qualification_var, font=("Arial", 11))

        row += add_row("Username", username_entry)
        row += add_row("Password", password_entry)
        row += add_row("Full Name", fullname_entry)
        row += add_row("Email", email_entry)
        row += add_row("Phone", phone_entry)
        row += add_row("Department", department_entry)
        row += add_row("Qualification", qualification_entry)

        # Status combobox
        tk.Label(form, text="Status", font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sticky="w
        status_combo = ttk.Combobox(form, textvariable=status_var, values=["Active", "Inactive"], state="rea
        status_combo.grid(row=row, column=1, padx=20, pady=8, sticky="w")
        row += 1

        def submit():
            username = username_entry.get().strip()
            password = password_entry.get().strip()
            fullname = fullname_entry.get().strip()
            email = email_entry.get().strip()
            phone = phone_entry.get().strip()
            department = department_entry.get().strip()
            qualification = qualification_entry.get().strip()
            status = status_var.get().strip()

            # Required fields
            required_map = {
                "username": username,
                "password": password,
                "full name": fullname,
                "email": email,
            }
            missing = [field for field, value in required_map.items() if not value]
            if missing:
                messagebox.showerror("Error", f"Please fill: {', '.join(missing)}.", parent=form)
                return

            # Normalize optionals
            phone = phone or None
            department = department or None
            qualification = qualification or None

            # Duplicates
            try:
                existing_user = db.check_username_exists(username)
            except Exception:
                existing_user = db.execute_query("SELECT user_id FROM users WHERE username = %s", (username,
            if existing_user:
                messagebox.showerror("Error", "Username already exists. Please choose another username.", pa
                return

            if email:
                existing_email = db.execute_query("SELECT teacher_id FROM teachers WHERE email = %s", (email
                if existing_email:
                    messagebox.showerror("Error", "Email already exists. Please use a different email.", par
                    return

            ok = db.add_teacher(username, password, fullname, email, phone, department, qualification, statu
            if ok:
                messagebox.showinfo("Success", "Teacher added successfully.", parent=form)
                self.load_teachers()
                self.load_dashboard_data()
                form.destroy()
            else:
                messagebox.showerror("Error", "Failed to add teacher. Username or email may already exist.",

        tk.Button(form, text="Add Teacher", font=("Arial", 12, "bold"), bg="#27ae60", fg="white", command=su

        form.transient(self.root)
        form.grab_set()
        self.root.wait_window(form)

    def edit_teacher(self):
```

```
        """Edit selected teacher"""
        selection = self.teachers_tree.selection()
        if not selection:
            messagebox.showwarning("Warning", "Please select a teacher to edit")
            return
        item = self.teachers_tree.item(selection[0])
        values = item['values']
        form = tk.Toplevel(self.root)
        form.title("Edit Teacher")
        form.geometry("520x460")
        form.resizable(False, False)

        # Pull fresh row from DB in case table is stale
        teacher_id = values[0]
        current = db.execute_query(
            "SELECT fullname, email, phone, department, qualification, status FROM teachers WHERE teacher_id
            (teacher_id,)
        )
        cur = current[0] if current else {}

        fullname_value = cur.get('fullname', values[1])
        email_value = cur.get('email', values[2])
        phone_value = cur.get('phone', values[3])
        department_value = cur.get('department', values[4])
        qualification_value = cur.get('qualification', values[5])
        status_var = tk.StringVar(value=cur.get('status', values[6]))

        row = 0
        def add_row(label_text, widget):
            tk.Label(form, text=label_text, font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sti
            widget.grid(row=row, column=1, padx=20, pady=8, sticky="w")
            return 1

        fullname_entry = tk.Entry(form, font=("Arial", 11))
        fullname_entry.insert(0, str(fullname_value or ""))
        row += add_row("Full Name", fullname_entry)

        email_entry = tk.Entry(form, font=("Arial", 11))
        email_entry.insert(0, str(email_value or ""))
        row += add_row("Email", email_entry)

        phone_entry = tk.Entry(form, font=("Arial", 11))
        phone_entry.insert(0, str(phone_value or ""))
        row += add_row("Phone", phone_entry)

        dept_entry = tk.Entry(form, font=("Arial", 11))
        dept_entry.insert(0, str(department_value or ""))
        row += add_row("Department", dept_entry)

        qual_entry = tk.Entry(form, font=("Arial", 11))
        qual_entry.insert(0, str(qualification_value or ""))
        row += add_row("Qualification", qual_entry)

        tk.Label(form, text="Status", font=("Arial", 11)).grid(row=row, column=0, padx=20, pady=8, sticky="w
        status_combo = ttk.Combobox(form, textvariable=status_var, values=["Active", "Inactive"], state="rea
        status_combo.grid(row=row, column=1, padx=20, pady=8, sticky="w")
        row += 1

        def submit():
            teacher_id = values[0]
            full_name = fullname_entry.get().strip()
            email = email_entry.get().strip()
            phone = phone_entry.get().strip()
            department = dept_entry.get().strip() or None
            qualification = qual_entry.get().strip() or None
            status = status_var.get().strip()
            if not full_name or not email or not phone:
                messagebox.showerror("Error", "Full name, email, and phone are required.", parent=form)
                return
            if db.update_teacher(teacher_id, full_name, email, phone, department, qualification, status):
                messagebox.showinfo("Success", "Teacher updated successfully.", parent=form)
                self.load_teachers()
                form.destroy()
            else:
                messagebox.showerror("Error", "Failed to update teacher.", parent=form)

        tk.Button(form, text="Update Teacher", font=("Arial", 12, "bold"), bg="#3498db", fg="white", command

        form.transient(self.root)
        form.grab_set()
        self.root.wait_window(form)

    def delete_teacher(self):
        """Delete selected teacher"""
        selection = self.teachers_tree.selection()
        if not selection:
```

```
            messagebox.showwarning("Warning", "Please select a teacher to delete")
            return

        if messagebox.askyesno("Confirm", "Are you sure you want to delete this teacher?"):
            item = self.teachers_tree.item(selection[0])
            teacher_id = item['values'][0]

            if db.delete_teacher(teacher_id):
                messagebox.showinfo("Success", "Teacher deleted successfully")
                self.load_teachers()
                self.load_dashboard_data()
            else:
                messagebox.showerror("Error", "Failed to delete teacher")

    def logout(self):
        """Logout and return to login"""
        if messagebox.askyesno("Logout", "Are you sure you want to logout?"):
            # Clean up event bindings
            self.root.unbind_all("<MouseWheel>")
            self.root.unbind_all("<Key>")
            self.root.destroy()
            # Relaunch a fresh login window in a new process to avoid Tk re-init and circular imports
            try:
                login_path = os.path.join(os.path.dirname(__file__), 'login.py')
                subprocess.Popen([sys.executable, login_path])
            except Exception as e:
                messagebox.showerror("Error", f"Failed to relaunch login: {e}")

    def on_closing(self):
        """Handle window closing"""
        # Clean up event bindings
        self.root.unbind_all("<MouseWheel>")
        self.root.unbind_all("<Key>")
        self.root.destroy()

    def load_students(self):
        """Load students data"""
        # Clear existing items
        for item in self.students_tree.get_children():
            self.students_tree.delete(item)

        students = db.get_all_students()
        for student in students:
            self.students_tree.insert('', 'end', values=(
                student['student_id'],
                student['fullname'],
                student['email'],
                student['phone'],
                student['gender'],
                student['status'],
                student['enrollment_date']
            ))

    def load_teachers(self):
        """Load teachers data"""
        # Clear existing items
        for item in self.teachers_tree.get_children():
            self.teachers_tree.delete(item)

        teachers = db.get_all_teachers()
        for teacher in teachers:
            self.teachers_tree.insert('', 'end', values=(
                teacher['teacher_id'],
                teacher['fullname'],
                teacher['email'],
                teacher['phone'],
                teacher['department'],
                teacher['qualification'],
                teacher['status']
            ))

    def filter_students(self, *args):
        """Filter students based on search"""
        search_term = self.student_search_var.get().lower()
        students = db.get_all_students()

        # Clear existing items
        for item in self.students_tree.get_children():
            self.students_tree.delete(item)

        # Filter and insert
        for student in students:
            if (search_term in student['fullname'].lower() or
                search_term in student['email'].lower()):
                self.students_tree.insert('', 'end', values=(
                    student['student_id'],
```

```
                student['fullname'],
                student['email'],
                student['phone'],
                student['gender'],
                student['status'],
                student['enrollment_date']
            ))

    def filter_teachers(self, *args):
        """Filter teachers based on search"""
        search_term = self.teacher_search_var.get().lower()
        teachers = db.get_all_teachers()

        # Clear existing items
        for item in self.teachers_tree.get_children():
            self.teachers_tree.delete(item)

        # Filter and insert
        for teacher in teachers:
            if (search_term in teacher['fullname'].lower() or
                search_term in teacher['email'].lower() or
                search_term in teacher['department'].lower()):
                self.teachers_tree.insert('', 'end', values=(
                    teacher['teacher_id'],
                    teacher['fullname'],
                    teacher['email'],
                    teacher['phone'],
                    teacher['department'],
                    teacher['qualification'],
                    teacher['status']
                ))
```

# Database Layer (database.py)

```
#!/usr/bin/env python3
"""
Database operations for Student Performance Monitoring System
"""

import mysql.connector
from mysql.connector import Error
import os
import hashlib
from tkinter import messagebox

class Database:
    def __init__(self):
        self.connection = None
        self.connect()

    def connect(self):
        """Establish connection to MySQL database"""
        try:
            # First connect to the server (without specifying database) to ensure DB exists
            self.connection = mysql.connector.connect(
                host='localhost',
                user='root',
                password='',
                autocommit=True,
                charset='utf8mb4'
            )
            if not self.connection or not self.connection.is_connected():
                raise Error("Unable to establish initial MySQL connection")

            cursor = self.connection.cursor()
            # Ensure database exists (utf8mb4)
            cursor.execute("CREATE DATABASE IF NOT EXISTS student_performance_db DEFAULT CHARACTER SET utf8m
            cursor.execute("USE student_performance_db")
            # Session settings
            try:
                cursor.execute("SET NAMES utf8mb4")
                cursor.execute("SET SESSION sql_mode = ''")
            except Exception:
                pass
            cursor.close()

            # Attempt to apply schema if not present
            try:
                self._ensure_schema()
            except Exception:
                # Non-fatal if schema file missing
                pass

            # Seed default subjects if missing
            try:
                self._seed_default_subjects()
            except Exception:
                pass

            print("[OK] Connected to MySQL database")
            return True
        except Error as e:
            print(f"[ERROR] Error connecting to MySQL: {e}")
            messagebox.showerror("Database Error",
                                 f"Failed to connect to database:\n{e}\n\nPlease ensure:\n"
                                 "1. XAMPP is running\n"
                                 "2. MySQL service is started\n"
                                 "3. Database 'student_performance_db' exists")
            return False
        return False

    def _ensure_schema(self):
        """Apply schema from database_setup.sql if available (idempotent)."""
        try:
            base_dir = os.path.dirname(os.path.abspath(__file__))
            schema_path = os.path.join(base_dir, 'database_setup.sql')
            if not os.path.exists(schema_path):
                return
            with open(schema_path, 'r', encoding='utf-8') as f:
                sql = f.read()
            # Execute script splitting on semicolons (simple best-effort)
            cursor = self.connection.cursor()
            statements = [s.strip() for s in sql.split(';') if s.strip()]
            for stmt in statements:
                try:
                    cursor.execute(stmt)
                except Exception as e:
```

```
                # Continue on benign errors (e.g., table exists)
                print(f"[WARN] Schema statement failed: {e}")
        cursor.close()
    except Exception as e:
        print(f"[WARN] Failed to apply schema: {e}")

def _seed_default_subjects(self):
    """Insert a set of common subjects if not present (idempotent)."""
    subjects = [
        ("Physics", "PHY102", 4, "General Physics"),
        ("Chemistry", "CHEM101", 4, "General Chemistry"),
        ("Informatics", "CS102", 4, "Computer Science Basics"),
        ("Biology", "BIO101", 4, "Introduction to Biology"),
        ("History", "HIS101", 3, "World History"),
        ("Geography", "GEO101", 3, "Geographical Studies"),
        ("Philosophy", "PHI101", 3, "Philosophical Thought"),
        ("Arabic", "ARB101", 3, "Arabic Language"),
        ("French", "FR101", 3, "French Language"),
        ("English", "ENG102", 3, "Advanced English"),
        ("Mathematics", "MATH102", 4, "Advanced Mathematics"),
    ]
    cursor = self.connection.cursor()
    # Use INSERT IGNORE to avoid duplicate subject_code errors
    cursor.execute("USE student_performance_db")
    for name, code, credits, desc in subjects:
        try:
            cursor.execute(
                "INSERT IGNORE INTO subjects (subject_name, subject_code, credits, description, teacher_
                (name, code, credits, desc)
            )
        except Exception as e:
            print(f"[WARN] Subject seed failed for {code}: {e}")
    cursor.close()

def execute_query(self, query, params=None):
    """Execute SELECT query and return results"""
    try:
        if not self.connection or not self.connection.is_connected():
            self.connect()
        cursor = self.connection.cursor(dictionary=True)
        cursor.execute(query, params or ())
        result = cursor.fetchall()
        cursor.close()
        return result
    except Error as e:
        print(f"[ERROR] Query error: {e} — attempting reconnect")
        try:
            self.connect()
            cursor = self.connection.cursor(dictionary=True)
            cursor.execute(query, params or ())
            result = cursor.fetchall()
            cursor.close()
            return result
        except Error as e2:
            print(f"[ERROR] Query retry failed: {e2}")
            return None

def execute_update(self, query, params=None):
    """Execute INSERT, UPDATE, DELETE query"""
    try:
        if not self.connection or not self.connection.is_connected():
            self.connect()
        cursor = self.connection.cursor()
        cursor.execute(query, params or ())
        self.connection.commit()
        cursor.close()
        return True
    except Error as e:
        print(f"[ERROR] Update error: {e} — attempting reconnect")
        try:
            self.connect()
            cursor = self.connection.cursor()
            cursor.execute(query, params or ())
            self.connection.commit()
            cursor.close()
            return True
        except Error as e2:
            print(f"[ERROR] Update retry failed: {e2}")
            return False

def hash_password(self, password):
    """Hash password using SHA-256"""
    return hashlib.sha256(password.encode()).hexdigest()

def verify_login(self, username, password):
    """Verify user login credentials"""
```

```
        hashed_password = self.hash_password(password)
        query = "SELECT * FROM users WHERE username = %s AND password = %s"
        result = self.execute_query(query, (username, hashed_password))
        return result[0] if result else None

    def get_user_by_id(self, user_id):
        """Get user by ID"""
        query = "SELECT * FROM users WHERE user_id = %s"
        result = self.execute_query(query, (user_id,))
        return result[0] if result else None

    def get_student_by_user_id(self, user_id):
        """Get student profile by user ID"""
        query = """
        SELECT s.*, u.username, u.role
        FROM students s
        JOIN users u ON s.user_id = u.user_id
        WHERE s.user_id = %s
        """
        result = self.execute_query(query, (user_id,))
        return result[0] if result else None

    def get_teacher_by_user_id(self, user_id):
        """Get teacher profile by user ID"""
        query = """
        SELECT t.*, u.username, u.role
        FROM teachers t
        JOIN users u ON t.user_id = u.user_id
        WHERE t.user_id = %s
        """
        result = self.execute_query(query, (user_id,))
        return result[0] if result else None

    def get_student_marks(self, student_id):
        """Get all marks for a student"""
        query = """
        SELECT m.*, s.subject_name, s.subject_code, s.credits, t.fullname as teacher_name
        FROM marks m
        JOIN subjects s ON m.subject_id = s.subject_id
        JOIN teachers t ON m.teacher_id = t.teacher_id
        WHERE m.student_id = %s
        ORDER BY m.exam_date DESC
        """
        return self.execute_query(query, (student_id,))

    def get_teacher_subjects(self, teacher_id):
        """Get subjects taught by a teacher"""
        query = "SELECT subject_id, subject_name FROM subjects WHERE teacher_id = %s ORDER BY subject_name"
        return self.execute_query(query, (teacher_id,)) or []

    def get_marks_for_teacher(self, teacher_id):
        """Get all marks entered by a teacher with student and subject names"""
        query = """
        SELECT
            m.mark_id,
            m.student_id,
            m.subject_id,
            st.fullname as student_name,
            sb.subject_name,
            m.marks_obtained,
            m.total_marks,
            m.exam_date
        FROM marks m
        JOIN students st ON m.student_id = st.student_id
        JOIN subjects sb ON m.subject_id = sb.subject_id
        WHERE m.teacher_id = %s
        ORDER BY m.exam_date DESC, m.mark_id DESC
        """
        return self.execute_query(query, (teacher_id,)) or []

    def get_teacher_students(self, teacher_id):
        """Get distinct students who have marks with this teacher"""
        query = """
        SELECT DISTINCT st.student_id, st.fullname, st.email, st.phone, st.gender, st.status
        FROM marks m
        JOIN students st ON m.student_id = st.student_id
        WHERE m.teacher_id = %s
        ORDER BY st.fullname
        """
        return self.execute_query(query, (teacher_id,)) or []

    def get_teacher_students_gender_counts(self, teacher_id):
        """Return gender counts for students taught by teacher (distinct students)"""
        query = """
        SELECT st.gender, COUNT(DISTINCT st.student_id) as count
        FROM marks m
```

```
        JOIN students st ON m.student_id = st.student_id
        WHERE m.teacher_id = %s
        GROUP BY st.gender
        """
        rows = self.execute_query(query, (teacher_id,)) or []
        data = { (r['gender'] or 'Other'): r['count'] for r in rows }
        return {
            'Male': data.get('Male', 0),
            'Female': data.get('Female', 0),
            'Other': data.get('Other', 0),
        }

    def get_teacher_subject_average_percentages(self, teacher_id, limit=10):
        """Average percentage per subject for a teacher"""
        query = """
        SELECT s.subject_name, AVG((m.marks_obtained / NULLIF(m.total_marks,0)) * 100) as avg_pct
        FROM marks m
        JOIN subjects s ON m.subject_id = s.subject_id
        WHERE m.teacher_id = %s
        GROUP BY s.subject_id, s.subject_name
        ORDER BY s.subject_name ASC
        LIMIT %s
        """
        return self.execute_query(query, (teacher_id, limit)) or []

    def get_teacher_monthly_trends_average(self, teacher_id, months=6):
        """Monthly average percentage for a teacher"""
        query = """
        SELECT DATE_FORMAT(m.exam_date, '%Y-%m') as ym,
               AVG((m.marks_obtained / NULLIF(m.total_marks,0)) * 100) as avg_pct
        FROM marks m
        WHERE m.teacher_id = %s AND m.exam_date IS NOT NULL
        GROUP BY ym
        ORDER BY ym DESC
        LIMIT %s
        """
        rows = self.execute_query(query, (teacher_id, months)) or []
        return list(reversed(rows))

    def add_mark(self, student_id, subject_id, teacher_id, marks_obtained, total_marks, exam_date):
        """Insert a new mark record"""
        query = (
            "INSERT INTO marks (student_id, subject_id, teacher_id, marks_obtained, total_marks, exam_date)
            "VALUES (%s, %s, %s, %s, %s, %s)"
        )
        return self.execute_update(query, (student_id, subject_id, teacher_id, marks_obtained, total_marks,

    def update_mark(self, mark_id, marks_obtained, total_marks, exam_date):
        """Update an existing mark record"""
        query = (
            "UPDATE marks SET marks_obtained = %s, total_marks = %s, exam_date = %s WHERE mark_id = %s"
        )
        return self.execute_update(query, (marks_obtained, total_marks, exam_date, mark_id))

    def delete_mark(self, mark_id):
        """Delete a mark record"""
        query = "DELETE FROM marks WHERE mark_id = %s"
        return self.execute_update(query, (mark_id,))

    def get_all_students(self):
        """Get all students"""
        query = """
        SELECT s.*, u.username, u.role
        FROM students s
        JOIN users u ON s.user_id = u.user_id
        ORDER BY s.fullname
        """
        return self.execute_query(query)

    def get_all_teachers(self):
        """Get all teachers"""
        query = """
        SELECT t.*, u.username, u.role
        FROM teachers t
        JOIN users u ON t.user_id = u.user_id
        ORDER BY t.fullname
        """
        return self.execute_query(query)

    def get_all_subjects(self):
        """Get all subjects"""
        query = """
        SELECT s.*, t.fullname as teacher_name
        FROM subjects s
        LEFT JOIN teachers t ON s.teacher_id = t.teacher_id
        ORDER BY s.subject_name
```

```
        """
        return self.execute_query(query)

    def add_student(self, username, password, fullname, email, phone, date_of_birth, gender, address, status
        """Add new student transactionally and return True on success."""
        try:
            if not self.connection or not self.connection.is_connected():
                self.connect()

            # Begin transaction (temporarily disable autocommit)
            prev_autocommit = self.connection.autocommit
            self.connection.autocommit = False
            cursor = self.connection.cursor()

            # Insert user
            hashed_password = self.hash_password(password)
            cursor.execute("INSERT INTO users (username, password, role) VALUES (%s, %s, 'student')", (usern
            user_id = cursor.lastrowid

            # Fallback if lastrowid isn't available for some reason
            if not user_id:
                cursor2 = self.connection.cursor()
                cursor2.execute("SELECT user_id FROM users WHERE username = %s ORDER BY user_id DESC LIMIT 1
                row = cursor2.fetchone()
                cursor2.close()
                user_id = row[0] if row else None

            if not user_id:
                raise Error("Failed to obtain user_id for new student user")

            # Insert student profile
            cursor.execute(
                """
                INSERT INTO students (user_id, fullname, email, phone, date_of_birth, gender, address, statu
                VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
                """,
                (user_id, fullname, email, phone, date_of_birth, gender, address, status)
            )

            # Commit both inserts
            self.connection.commit()
            cursor.close()
            return True
        except Error as e:
            try:
                if self.connection and self.connection.is_connected():
                    self.connection.rollback()
            except Exception:
                pass
            print(f"[ERROR] Error adding student: {e}")
            return False
        finally:
            try:
                # Restore autocommit
                if self.connection and self.connection.is_connected():
                    self.connection.autocommit = True
            except Exception:
                pass

    def add_teacher(self, username, password, fullname, email, phone, department, qualification, status):
        """Add new teacher transactionally and return True on success."""
        try:
            if not self.connection or not self.connection.is_connected():
                self.connect()

            # Begin transaction (temporarily disable autocommit)
            prev_autocommit = self.connection.autocommit
            self.connection.autocommit = False
            cursor = self.connection.cursor()

            # Insert user
            hashed_password = self.hash_password(password)
            cursor.execute("INSERT INTO users (username, password, role) VALUES (%s, %s, 'teacher')", (usern
            user_id = cursor.lastrowid

            # Fallback if lastrowid isn't available
            if not user_id:
                cursor2 = self.connection.cursor()
                cursor2.execute("SELECT user_id FROM users WHERE username = %s ORDER BY user_id DESC LIMIT 1
                row = cursor2.fetchone()
                cursor2.close()
                user_id = row[0] if row else None

            if not user_id:
                raise Error("Failed to obtain user_id for new teacher user")
            # Insert teacher profile
```

```
            cursor.execute(
                """
                INSERT INTO teachers (user_id, fullname, email, phone, department, qualification, status)
                VALUES (%s, %s, %s, %s, %s, %s, %s)
                """,
                (user_id, fullname, email, phone, department, qualification, status)
            )

            # Commit both inserts
            self.connection.commit()
            cursor.close()
            return True
        except Error as e:
            try:
                if self.connection and self.connection.is_connected():
                    self.connection.rollback()
            except Exception:
                pass
            print(f"[ERROR] Error adding teacher: {e}")
            return False
        finally:
            try:
                # Restore autocommit
                if self.connection and self.connection.is_connected():
                    self.connection.autocommit = True
            except Exception:
                pass

    def update_student(self, student_id, fullname, email, phone, date_of_birth, gender, address, status):
        """Update student information"""
        query = """
        UPDATE students
        SET fullname = %s, email = %s, phone = %s, date_of_birth = %s, gender = %s, address = %s, status = %
        WHERE student_id = %s
        """
        return self.execute_update(query, (fullname, email, phone, date_of_birth, gender, address, status, s

    def update_teacher(self, teacher_id, fullname, email, phone, department, qualification, status):
        """Update teacher information"""
        query = """
        UPDATE teachers
        SET fullname = %s, email = %s, phone = %s, department = %s, qualification = %s, status = %s
        WHERE teacher_id = %s
        """
        return self.execute_update(query, (fullname, email, phone, department, qualification, status, teache

    def delete_student(self, student_id):
        """Delete student (cascades to user and marks)"""
        query = "DELETE FROM students WHERE student_id = %s"
        return self.execute_update(query, (student_id,))

    def delete_teacher(self, teacher_id):
        """Delete teacher (cascades to user)"""
        query = "DELETE FROM teachers WHERE teacher_id = %s"
        return self.execute_update(query, (teacher_id,))

    def get_system_stats(self):
        """Get system statistics for admin dashboard"""
        stats = {}

        # Count students
        result = self.execute_query("SELECT COUNT(*) as count FROM students WHERE status = 'Active'")
        stats['active_students'] = result[0]['count'] if result else 0

        # Count teachers
        result = self.execute_query("SELECT COUNT(*) as count FROM teachers WHERE status = 'Active'")
        stats['active_teachers'] = result[0]['count'] if result else 0

        # Count subjects
        result = self.execute_query("SELECT COUNT(*) as count FROM subjects")
        stats['total_subjects'] = result[0]['count'] if result else 0

        # Average percentage across all marks (marks_obtained / total_marks * 100)
        result = self.execute_query("SELECT AVG((marks_obtained / NULLIF(total_marks,0)) * 100) as avg_pct F
        stats['average_marks'] = round(result[0]['avg_pct'], 2) if result and result[0]['avg_pct'] else 0

        return stats

    def get_gender_distribution(self):
        """Return counts by gender for students"""
        query = "SELECT gender, COUNT(*) as count FROM students GROUP BY gender"
        rows = self.execute_query(query) or []
        data = { (row['gender'] or 'Other'): row['count'] for row in rows }
        return {
            'Male': data.get('Male', 0),
            'Female': data.get('Female', 0),
```

```
            'Other': data.get('Other', 0),
        }

    def get_subject_average_percentages(self, limit=8):
        """Average percentage per subject, top N subjects by name"""
        query = (
            """
            SELECT s.subject_name, AVG((m.marks_obtained / NULLIF(m.total_marks,0)) * 100) as avg_pct
            FROM marks m
            JOIN subjects s ON m.subject_id = s.subject_id
            GROUP BY s.subject_id, s.subject_name
            ORDER BY s.subject_name ASC
            LIMIT %s
            """
        )
        return self.execute_query(query, (limit,)) or []

    def get_monthly_trends_average(self, months=6):
        """Average percentage per recent month (YYYY-MM)"""
        query = (
            """
            SELECT DATE_FORMAT(exam_date, '%Y-%m') as ym,
                   AVG((marks_obtained / NULLIF(total_marks,0)) * 100) as avg_pct
            FROM marks
            WHERE exam_date IS NOT NULL
            GROUP BY ym
            ORDER BY ym DESC
            LIMIT %s
            """
        )
        rows = self.execute_query(query, (months,)) or []
        return list(reversed(rows))

    def get_top_students(self, limit=3):
        """Get top performing students with their CGPA"""
        query = """
        SELECT
            s.student_id,
            s.fullname,
            s.email,
            s.gender,
            COUNT(m.mark_id) as total_exams,
            AVG((m.marks_obtained / m.total_marks) * 100) as avg_percentage,
            CASE
                WHEN AVG((m.marks_obtained / m.total_marks) * 100) >= 90 THEN 4.0
                WHEN AVG((m.marks_obtained / m.total_marks) * 100) >= 80 THEN 3.5
                WHEN AVG((m.marks_obtained / m.total_marks) * 100) >= 70 THEN 3.0
                WHEN AVG((m.marks_obtained / m.total_marks) * 100) >= 60 THEN 2.5
                WHEN AVG((m.marks_obtained / m.total_marks) * 100) >= 50 THEN 2.0
                ELSE 1.0
            END as cgpa
        FROM students s
        JOIN marks m ON s.student_id = m.student_id
        WHERE s.status = 'Active'
        GROUP BY s.student_id, s.fullname, s.email, s.gender
        HAVING COUNT(m.mark_id) > 0
        ORDER BY avg_percentage DESC
        LIMIT %s
        """
        return self.execute_query(query, (limit,))

    # Password reset functionality
    def check_username_exists(self, username):
        """Check if username exists in the database"""
        query = "SELECT user_id, username, role FROM users WHERE username = %s"
        result = self.execute_query(query, (username,))
        return result[0] if result else None

    def update_password(self, username, new_password):
        """Update user password"""
        hashed_password = self.hash_password(new_password)
        query = "UPDATE users SET password = %s WHERE username = %s"
        return self.execute_update(query, (hashed_password, username))

    def get_user_email(self, username):
        """Get user email for password reset"""
        query = """
        SELECT u.username, u.role,
               COALESCE(s.email, t.email) as email
        FROM users u
        LEFT JOIN students s ON u.user_id = s.user_id
        LEFT JOIN teachers t ON u.user_id = t.user_id
        WHERE u.username = %s
        """
        result = self.execute_query(query, (username,))
        return result[0] if result else None
```

```
    def close(self):
        """Close database connection"""
        if self.connection and self.connection.is_connected():
            self.connection.close()
            print("[OK] Database connection closed")

# Global database instance
db = Database()
```

# Login Window (login.py)

```
#!/usr/bin/env python3
"""
Login window for Student Performance Monitoring System
"""

import tkinter as tk
from tkinter import ttk, messagebox
from database import db
import admin
import teacher
import student
import io
import random
import string
from PIL import Image, ImageTk
try:
    # Optional dependency; present in requirements.txt
    from captcha.image import ImageCaptcha
except Exception:
    ImageCaptcha = None
try:
    from tktooltip import ToolTip as Tooltip
except Exception:
    Tooltip = None

class LoginWindow:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Student Performance Monitoring System - Login")
        self.root.geometry("900x600")
        self.root.resizable(False, False)

        # Center the window
        self.center_window()

        # Configure grid weights
        self.root.grid_columnconfigure(0, weight=1)
        self.root.grid_columnconfigure(1, weight=1)
        self.root.grid_rowconfigure(0, weight=1)

        # Create left panel (illustration)
        self.create_left_panel()

        # Create right panel (login form)
        self.create_right_panel()

        # Bind Enter key to login
        self.root.bind('<Return>', lambda e: self.login())

        # Focus on username entry
        self.username_entry.focus()

    def create_left_panel(self):
        """Create left panel with illustration"""
        left_frame = tk.Frame(self.root, bg="#f8f9fa")
        left_frame.grid(row=0, column=0, sticky="nsew", padx=0, pady=0)

        # Create illustration frame
        illustration_frame = tk.Frame(left_frame, bg="#f8f9fa")
        illustration_frame.pack(expand=True, fill="both", padx=40, pady=40)

        # Title for left panel
        title_label = tk.Label(illustration_frame,
                               text="Student Performance\nMonitoring System",
                               font=("Arial", 24, "bold"),
                               fg="#2c3e50",
                               bg="#f8f9fa",
                               justify="center")
        title_label.pack(pady=(50, 30))

        # Subtitle
        subtitle_label = tk.Label(illustration_frame,
                                  text="Track, Analyze, and Improve\nAcademic Performance",
                                  font=("Arial", 14),
                                  fg="#7f8c8d",
                                  bg="#f8f9fa",
                                  justify="center")
        subtitle_label.pack(pady=(0, 50))

        # Create a simple illustration using shapes
        canvas = tk.Canvas(illustration_frame, width=300, height=200,
                           bg="#f8f9fa", highlightthickness=0)
        canvas.pack(pady=20)
```

```
        # Draw a simple desk illustration
        # Desk
        canvas.create_rectangle(50, 120, 250, 140, fill="#8b4513", outline="")
        # Chair
        canvas.create_rectangle(60, 100, 90, 120, fill="#654321", outline="")
        canvas.create_rectangle(70, 80, 80, 100, fill="#654321", outline="")
        # Laptop
        canvas.create_rectangle(80, 90, 180, 110, fill="#2c3e50", outline="")
        canvas.create_rectangle(85, 95, 175, 105, fill="#ecf0f1", outline="")
        # Lamp
        canvas.create_oval(200, 70, 220, 90, fill="#f39c12", outline="")
        canvas.create_line(210, 90, 210, 120, fill="#95a5a6", width=3)
        # Plant
        canvas.create_oval(240, 100, 260, 120, fill="#27ae60", outline="")
        canvas.create_rectangle(245, 120, 255, 130, fill="#8b4513", outline="")

        # Features list
        features_frame = tk.Frame(illustration_frame, bg="#f8f9fa")
        features_frame.pack(pady=30)

        features = [
            "■ Real-time Performance Tracking",
            "■ Interactive Analytics & Charts",
            "■ Multi-role Access (Admin/Teacher/Student)",
            "■ Secure Authentication System"
        ]

        for feature in features:
            feature_label = tk.Label(features_frame,
                                     text=feature,
                                     font=("Arial", 12),
                                     fg="#34495e",
                                     bg="#f8f9fa",
                                     anchor="w")
            feature_label.pack(pady=5, anchor="w")

    def create_right_panel(self):
        """Create right panel with login form"""
        right_frame = tk.Frame(self.root, bg="white")
        right_frame.grid(row=0, column=1, sticky="nsew", padx=0, pady=0)

        # Main login container
        login_container = tk.Frame(right_frame, bg="white")
        login_container.pack(expand=True, fill="both", padx=60, pady=40)

        # Login title
        login_title = tk.Label(login_container,
                               text="Login",
                               font=("Arial", 28, "bold"),
                               fg="#2c3e50",
                               bg="white")
        login_title.pack(pady=(0, 40))

        # Username field
        username_frame = tk.Frame(login_container, bg="white")
        username_frame.pack(fill="x", pady=10)

        username_label = tk.Label(username_frame,
                                  text="username",
                                  font=("Arial", 12, "bold"),
                                  fg="#2c3e50",
                                  bg="white",
                                  anchor="w")
        username_label.pack(anchor="w", pady=(0, 5))

        # Username entry with icon
        username_entry_frame = tk.Frame(username_frame, bg="white", relief="solid", bd=1)
        username_entry_frame.pack(fill="x", pady=(0, 10))

        # User icon (simple text representation)
        user_icon = tk.Label(username_entry_frame,
                             text="■",
                             font=("Arial", 14),
                             bg="white",
                             fg="#7f8c8d")
        user_icon.pack(side="left", padx=10)

        self.username_var = tk.StringVar()
        self.username_entry = tk.Entry(username_entry_frame,
                                       textvariable=self.username_var,
                                       font=("Arial", 12),
                                       relief="flat",
                                       bg="white",
                                       fg="#2c3e50",
                                       insertbackground="#2c3e50")
        self.username_entry.pack(side="left", fill="x", expand=True, padx=(0, 10), pady=10)
```

```
# Password field
password_frame = tk.Frame(login_container, bg="white")
password_frame.pack(fill="x", pady=10)

password_label = tk.Label(password_frame,
                          text="Password",
                          font=("Arial", 12, "bold"),
                          fg="#2c3e50",
                          bg="white",
                          anchor="w")
password_label.pack(anchor="w", pady=(0, 5))

# Password entry with icon
password_entry_frame = tk.Frame(password_frame, bg="white", relief="solid", bd=1)
password_entry_frame.pack(fill="x", pady=(0, 10))

# Lock icon
lock_icon = tk.Label(password_entry_frame,
                     text="■",
                     font=("Arial", 14),
                     bg="white",
                     fg="#7f8c8d")
lock_icon.pack(side="left", padx=10)

self.password_var = tk.StringVar()
self.password_entry = tk.Entry(password_entry_frame,
                               textvariable=self.password_var,
                               font=("Arial", 12),
                               relief="flat",
                               bg="white",
                               fg="#2c3e50",
                               show="•",
                               insertbackground="#2c3e50")
self.password_entry.pack(side="left", fill="x", expand=True, padx=(0, 10), pady=10)

# Eye icon for password visibility
self.show_password = tk.BooleanVar()
eye_icon = tk.Label(password_entry_frame,
                    text="■■",
                    font=("Arial", 12),
                    bg="white",
                    fg="#7f8c8d",
                    cursor="hand2")
eye_icon.pack(side="right", padx=10)
eye_icon.bind("<Button-1>", self.toggle_password_visibility)

# CAPTCHA
captcha_frame = tk.Frame(login_container, bg="white")
captcha_frame.pack(fill="x", pady=10)
captcha_label = tk.Label(captcha_frame, text="Captcha", font=("Arial", 12, "bold"), fg="#2c3e50", bg
captcha_label.pack(anchor="w", pady=(0,5))
captcha_row = tk.Frame(captcha_frame, bg="white")
captcha_row.pack(fill="x")

self.captcha_var = tk.StringVar()
self.captcha_entry = tk.Entry(captcha_row, textvariable=self.captcha_var, font=("Arial", 12), relief
self.captcha_entry.pack(side="left", padx=(0,10))

self.captcha_image_label = tk.Label(captcha_row, bg="white")
self.captcha_image_label.pack(side="left")
self.captcha_image_label.bind("<Button-1>", lambda e: self.refresh_captcha())
refresh_btn = tk.Button(captcha_row,
                        text="■",
                        command=self.refresh_captcha,
                        relief="raised",
                        bg="#ecf0f1",
                        fg="#2c3e50",
                        width=4,
                        height=2,
                        font=("Arial", 14, "bold"),
                        cursor="hand2",
                        bd=1)
refresh_btn.pack(side="right", padx=10, pady=2)
if Tooltip is not None:
    Tooltip(refresh_btn, "Refresh Captcha (Alt+R)")
# Keyboard shortcut
self.root.bind_all("<Alt-r>", lambda e: self.refresh_captcha())

self.refresh_captcha()

# Forgot password link
forgot_frame = tk.Frame(login_container, bg="white")
forgot_frame.pack(fill="x", pady=5)

forgot_link = tk.Label(forgot_frame,
                       text="Forgot Password?",
```

```
                        font=("Arial", 11),
                        fg="#3498db",
                        bg="white",
                        cursor="hand2")
        forgot_link.pack(anchor="e")
        try:
            import forgot_password  # local module; loaded on demand
            forgot_link.bind("<Button-1>", lambda e: forgot_password.ForgotPasswordWindow(self.root))
        except Exception:
            forgot_link.bind("<Button-1>", lambda e: messagebox.showinfo("Info", "Contact your administrator

        # Rounded Login button (full width)
        self.create_rounded_button(login_container, text="Login", command=self.login)

        # Demo credentials
        demo_frame = tk.Frame(login_container, bg="white")
        demo_frame.pack(fill="x", pady=20)

        demo_title = tk.Label(demo_frame,
                              text="Demo Credentials",
                              font=("Arial", 12, "bold"),
                              fg="#2c3e50",
                              bg="white")
        demo_title.pack(pady=(0, 10))

        demo_credentials = [
            "■■■ Admin: admin / admin123",
            "■■■ Teacher: teacher1 / teacher123",
            "■■■ Student: student1 / student123"
        ]

        for credential in demo_credentials:
            cred_label = tk.Label(demo_frame,
                                  text=credential,
                                  font=("Arial", 10),
                                  fg="#7f8c8d",
                                  bg="white",
                                  anchor="w")
            cred_label.pack(anchor="w", pady=2)

        # Get started link
        get_started_frame = tk.Frame(login_container, bg="white")
        get_started_frame.pack(fill="x", pady=20)

        get_started_text = tk.Label(get_started_frame,
                                    text="Don't have an account? ",
                                    font=("Arial", 11),
                                    fg="#7f8c8d",
                                    bg="white")
        get_started_text.pack(side="left")

        get_started_link = tk.Label(get_started_frame,
                                    text="Get Started For Free",
                                    font=("Arial", 11, "bold"),
                                    fg="#3498db",
                                    bg="white",
                                    cursor="hand2")
        get_started_link.pack(side="left")
        get_started_link.bind("<Button-1>", lambda e: messagebox.showinfo("Info", "Contact your administrato

    def create_rounded_button(self, parent, text, command):
        """Create a full-width rounded gold button similar to the screenshot."""
        button_height = 72
        radius = 30
        bg_color = "#c88f05"  # gold-like
        bg_hover = "#d9a31a"
        text_color = "#111111"

        canvas = tk.Canvas(parent, height=button_height, bg="white", highlightthickness=0)
        canvas.pack(fill="x", pady=5, side="bottom")

        def draw(width):
            canvas.delete("all")
            pad = 80
            x1, y1 = pad, 8
            x2, y2 = width - pad, button_height - 8
            r = radius
            # rounded rectangle using polygons + arcs
            canvas.create_arc(x1, y1, x1+2*r, y1+2*r, start=90, extent=180, fill=bg_color, outline=bg_color)
            canvas.create_arc(x2-2*r, y1, x2, y1+2*r, start=270, extent=180, fill=bg_color, outline=bg_color
            canvas.create_rectangle(x1+r, y1, x2-r, y2, fill=bg_color, outline=bg_color)
            canvas.create_rectangle(x1, y1+r, x2, y2-r, fill=bg_color, outline=bg_color)
            canvas.create_text((width)//2, button_height//2, text=text, font=("Arial", 20, "bold"), fill=tex

        # Force initial draw after Canvas is realized
        canvas.after(50, lambda: draw(canvas.winfo_width()))
```

```
        def on_resize(event):
            draw(event.width)

        def on_click(_):
            if callable(command):
                command()

        def on_enter(_):
            nonlocal bg_color
            old = bg_color
            bg_color = bg_hover
            draw(canvas.winfo_width())
            bg_color = old

        def on_leave(_):
            draw(canvas.winfo_width())

        canvas.bind("<Configure>", on_resize)
        canvas.bind("<Button-1>", on_click)
        canvas.bind("<Enter>", on_enter)
        canvas.bind("<Leave>", on_leave)

    def _generate_captcha_text(self, length=5):
        alphabet = string.ascii_uppercase + string.digits
        return ''.join(random.choice(alphabet) for _ in range(length))

    def refresh_captcha(self):
        """Generate and display a new captcha image"""
        self.captcha_answer = self._generate_captcha_text()
        if ImageCaptcha is not None:
            generator = ImageCaptcha(width=160, height=60)
            image = generator.generate_image(self.captcha_answer)
        else:
            # Fallback: render plain text with Pillow
            image = Image.new('RGB', (160, 60), color=(255, 255, 255))
            from PIL import ImageDraw
            draw = ImageDraw.Draw(image)
            draw.text((10, 15), self.captcha_answer, fill=(0, 0, 0))
        buffer = io.BytesIO()
        image.save(buffer, format='PNG')
        buffer.seek(0)
        pil_img = Image.open(buffer)
        # Bind the image to this window's Tk instance and keep strong references
        self._tk_captcha = ImageTk.PhotoImage(pil_img, master=self.root)
        try:
            self.captcha_image_label.configure(image=self._tk_captcha)
            # Prevent garbage collection by storing on the label as well
            self.captcha_image_label.image = self._tk_captcha
        except tk.TclError:
            # Window might be closing; safely ignore
            return
        self.captcha_var.set("")

    def toggle_password_visibility(self, event):
        """Toggle password visibility"""
        if self.password_entry.cget('show') == '•':
            self.password_entry.config(show='')
        else:
            self.password_entry.config(show='•')

    def center_window(self):
        """Center the window on screen"""
        self.root.update_idletasks()
        width = self.root.winfo_width()
        height = self.root.winfo_height()
        x = (self.root.winfo_screenwidth() // 2) - (width // 2)
        y = (self.root.winfo_screenheight() // 2) - (height // 2)
        self.root.geometry(f'{width}x{height}+{x}+{y}')

    def login(self):
        """Handle login authentication"""
        username = self.username_var.get().strip()
        password = self.password_var.get().strip()
        captcha_input = self.captcha_var.get().strip().upper()

        if not username or not password:
            messagebox.showerror("Error", "Please enter both username and password")
            return

        if not captcha_input:
            messagebox.showerror("Error", "Please enter the captcha")
            return
        if captcha_input != getattr(self, 'captcha_answer', ''):
            messagebox.showerror("Error", "Incorrect captcha. Please try again.")
            self.refresh_captcha()
            return
```

```
        # Authenticate user
        user = db.verify_login(username, password)

        if user:
            self.root.withdraw()  # Hide login window
            try:
                # Open appropriate dashboard based on role
                if user['role'] == 'admin':
                    admin.AdminDashboard(user).run()
                elif user['role'] == 'teacher':
                    teacher_profile = db.get_teacher_by_user_id(user['user_id'])
                    if teacher_profile:
                        teacher.TeacherDashboard(user, teacher_profile)
                    else:
                        messagebox.showerror("Error", "Teacher profile not found")
                        self.root.deiconify()
                elif user['role'] == 'student':
                    student_profile = db.get_student_by_user_id(user['user_id'])
                    if student_profile:
                        student.StudentDashboard(user, student_profile)
                    else:
                        messagebox.showerror("Error", "Student profile not found")
                        self.root.deiconify()
            except Exception as e:
                # Surface dashboard errors gracefully and return to login
                messagebox.showerror("Dashboard Error", f"Failed to open dashboard:\n{e}")
                self.root.deiconify()
        else:
            messagebox.showerror("Error", "Invalid username or password")
            self.password_var.set("")
            self.password_entry.focus()
            self.refresh_captcha()

    def run(self):
        """Start the login window"""
        self.root.mainloop()

if __name__ == "__main__":
    login_window = LoginWindow()
    login_window.run()
```

# Database Schema (database_setup.sql)

```
-- ==============================
-- Create database
-- ==============================
CREATE DATABASE IF NOT EXISTS student_performance_db;
USE student_performance_db;

-- ==============================
-- 1. Users table
-- ==============================
CREATE TABLE IF NOT EXISTS users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    role ENUM('admin', 'teacher', 'student') NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- ==============================
-- 2. Students table
-- ==============================
CREATE TABLE IF NOT EXISTS students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    fullname VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(20),
    date_of_birth DATE,
    gender ENUM('Male', 'Female', 'Other'),
    address TEXT,
    enrollment_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('Active', 'Inactive', 'Graduated') DEFAULT 'Active',
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);

-- ==============================
-- 3. Teachers table
-- ==============================
CREATE TABLE IF NOT EXISTS teachers (
    teacher_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    fullname VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(20),
    department VARCHAR(50),
    qualification VARCHAR(100),
    hire_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status ENUM('Active', 'Inactive') DEFAULT 'Active',
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);

-- ==============================
-- 4. Subjects table
-- ==============================
CREATE TABLE IF NOT EXISTS subjects (
    subject_id INT AUTO_INCREMENT PRIMARY KEY,
    subject_name VARCHAR(100) NOT NULL,
    subject_code VARCHAR(20) UNIQUE NOT NULL,
    credits INT DEFAULT 3,
    description TEXT,
    teacher_id INT NULL,
    FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id) ON DELETE SET NULL
);

-- ==============================
-- 5. Marks table
-- ==============================
CREATE TABLE IF NOT EXISTS marks (
    mark_id INT AUTO_INCREMENT PRIMARY KEY,
    student_id INT,
    subject_id INT,
    teacher_id INT,
    exam_type ENUM('Quiz', 'Midterm', 'Final', 'Assignment', 'Project') NOT NULL,
    marks_obtained DECIMAL(5,2) NOT NULL,
    total_marks DECIMAL(5,2) DEFAULT 100.00,
    exam_date DATE,
    semester VARCHAR(20),
    academic_year VARCHAR(10),
    remarks TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (student_id) REFERENCES students(student_id) ON DELETE CASCADE,
    FOREIGN KEY (subject_id) REFERENCES subjects(subject_id) ON DELETE CASCADE,
    FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id) ON DELETE CASCADE,
    CONSTRAINT chk_marks CHECK (marks_obtained <= total_marks)
```

```
);

-- ==============================
-- Insert sample data
-- ==============================

-- Sample users (passwords are SHA-256 hashed for demo)
INSERT INTO users (username, password, role) VALUES
('admin', '240be518fabd2724ddb6f04eeb1da5967448d7e831c08c8fa822809f74c720a9', 'admin'),
('teacher1', 'cde383eee8ee7a4400adf7a15f716f179a2eb97646b37e089eb8d6d04e663416', 'teacher'),
('teacher2', 'cde383eee8ee7a4400adf7a15f716f179a2eb97646b37e089eb8d6d04e663416', 'teacher'),
('student1', '703b0a3d6ad75b649a28adde7d83c6251da457549263bc7ff45ec709b0a8448b', 'student'),
('student2', '703b0a3d6ad75b649a28adde7d83c6251da457549263bc7ff45ec709b0a8448b', 'student'),
('student3', '703b0a3d6ad75b649a28adde7d83c6251da457549263bc7ff45ec709b0a8448b', 'student');

-- Sample teachers
INSERT INTO teachers (user_id, fullname, email, phone, department, qualification) VALUES
(2, 'John Smith', 'john.smith@school.com', '123-456-7890', 'Mathematics', 'M.Sc. Mathematics'),
(3, 'Sarah Johnson', 'sarah.johnson@school.com', '123-456-7891', 'Science', 'Ph.D. Physics');

-- Sample students
INSERT INTO students (user_id, fullname, email, phone, date_of_birth, gender, address) VALUES
(4, 'Alice Brown', 'alice.brown@student.com', '123-456-7892', '2005-03-15', 'Female', '123 Student St'),
(5, 'Bob Wilson', 'bob.wilson@student.com', '123-456-7893', '2005-07-22', 'Male', '456 Student Ave'),
(6, 'Carol Davis', 'carol.davis@student.com', '123-456-7894', '2005-11-08', 'Female', '789 Student Blvd');

-- Sample subjects
INSERT INTO subjects (subject_name, subject_code, credits, description, teacher_id) VALUES
('Mathematics', 'MATH101', 4, 'Advanced Mathematics', 1),
('Physics', 'PHY101', 4, 'Introduction to Physics', 2),
('English', 'ENG101', 3, 'English Literature', 1),
('Computer Science', 'CS101', 4, 'Programming Fundamentals', 2);

-- Sample marks
INSERT INTO marks (student_id, subject_id, teacher_id, exam_type, marks_obtained, total_marks, exam_date, se
(1, 1, 1, 'Midterm', 85.5, 100.0, '2024-03-15', 'Spring 2024', '2024'),
(1, 1, 1, 'Final', 92.0, 100.0, '2024-05-20', 'Spring 2024', '2024'),
(1, 2, 2, 'Midterm', 78.0, 100.0, '2024-03-20', 'Spring 2024', '2024'),
(1, 2, 2, 'Final', 88.5, 100.0, '2024-05-25', 'Spring 2024', '2024'),
(2, 1, 1, 'Midterm', 90.0, 100.0, '2024-03-15', 'Spring 2024', '2024'),
(2, 1, 1, 'Final', 95.5, 100.0, '2024-05-20', 'Spring 2024', '2024'),
(2, 3, 1, 'Midterm', 82.0, 100.0, '2024-03-18', 'Spring 2024', '2024'),
(3, 1, 1, 'Midterm', 75.5, 100.0, '2024-03-15', 'Spring 2024', '2024'),
(3, 4, 2, 'Midterm', 88.0, 100.0, '2024-03-22', 'Spring 2024', '2024'),
(3, 4, 2, 'Final', 91.0, 100.0, '2024-05-28', 'Spring 2024', '2024');
```

# ML Model Utilities (ml_model.py)

```
#!/usr/bin/env python3
"""
ML model utilities: train and predict student performance percentages.
"""

import os, math, datetime as dt
import pandas as pd
import joblib
from sklearn.ensemble import RandomForestRegressor
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import Pipeline
from database import db

MODEL_PATH = os.path.join(os.path.dirname(__file__), "models", "grade_predictor.joblib")

def _fetch_marks_df():
    rows = db.execute_query(
        """
        SELECT m.student_id, m.subject_id, m.teacher_id, m.exam_date,
               m.marks_obtained, m.total_marks
        FROM marks m
        WHERE m.total_marks IS NOT NULL AND m.marks_obtained IS NOT NULL
        ORDER BY m.student_id, m.subject_id, m.exam_date
        """
    ) or []
    if not rows:
        return pd.DataFrame([])
    df = pd.DataFrame(rows)
    df["exam_date"] = pd.to_datetime(df["exam_date"], errors="coerce")
    df["pct"] = (df["marks_obtained"].astype(float) / df["total_marks"].astype(float)) * 100.0
    return df.dropna(subset=["pct"])

def _feature_engineer(df: pd.DataFrame) -> pd.DataFrame:
    if df.empty:
        return df
    df = df.sort_values(["student_id", "subject_id", "exam_date"])
    # Overall student expanding mean (shifted)
    df["student_overall_avg"] = (
        df.groupby("student_id")["pct"].apply(lambda s: s.expanding().mean().shift(1)).reset_index(level=0, 
    )
    # Per student-subject expanding mean (shifted)
    df["subj_avg"] = (
        df.groupby(["student_id","subject_id"])["pct"].apply(lambda s: s.expanding().mean().shift(1)).reset_
    )
    # Attempt count and recency
    df["attempts_subj"] = df.groupby(["student_id","subject_id"]).cumcount()
    prev_date = df.groupby(["student_id","subject_id"])["exam_date"].shift(1)
    df["days_since"] = (df["exam_date"] - prev_date).dt.days.fillna(60)
    # Fills
    df["student_overall_avg"] = df["student_overall_avg"].fillna(df["pct"].mean())
    df["subj_avg"] = df["subj_avg"].fillna(df["student_overall_avg"])
    df["days_since"] = df["days_since"].fillna(60)
    return df

def build_dataset():
    df = _fetch_marks_df()
    if df.empty:
        return None, None, None
    df = _feature_engineer(df)
    df = df.dropna(subset=["student_overall_avg","subj_avg"])  # need history
    features = ["student_overall_avg","subj_avg","attempts_subj","days_since","subject_id","teacher_id"]
    X = df[features].copy()
    y = df["pct"].astype(float)
    return X, y, features

def train_and_save():
    X, y, features = build_dataset()
    if X is None or len(X) < 20:
        print("[WARN] Not enough data to train")
        return False
    preproc = ColumnTransformer([
        ("num", StandardScaler(), ["student_overall_avg","subj_avg","attempts_subj","days_since"]),
        ("cat", OneHotEncoder(handle_unknown="ignore"), ["subject_id","teacher_id"]),
    ])
    model = Pipeline([
        ("prep", preproc),
        ("rf", RandomForestRegressor(n_estimators=200, random_state=42))
    ])
    model.fit(X, y)
    os.makedirs(os.path.join(os.path.dirname(__file__), "models"), exist_ok=True)
    joblib.dump({"model": model, "features": features}, MODEL_PATH)
    print(f"[OK] Saved model to {MODEL_PATH}")
```

```
    return True

def load_model():
    if not os.path.exists(MODEL_PATH):
        train_and_save()
    if not os.path.exists(MODEL_PATH):
        return None
    return joblib.load(MODEL_PATH)

def _latest_stats(student_id, subject_id):
    rows = db.execute_query(
        """
        SELECT m.student_id, m.subject_id, m.teacher_id, m.exam_date,
               m.marks_obtained, m.total_marks
        FROM marks m
        WHERE m.student_id = %s AND m.subject_id = %s AND m.total_marks IS NOT NULL AND m.marks_obtained IS
        ORDER BY m.exam_date
        """, (student_id, subject_id)
    ) or []
    if not rows:
        return None, None
    df = pd.DataFrame(rows)
    df["exam_date"] = pd.to_datetime(df["exam_date"], errors="coerce")
    df["pct"] = (df["marks_obtained"].astype(float) / df["total_marks"].astype(float)) * 100.0
    all_student = db.execute_query(
        """
        SELECT m.exam_date, m.marks_obtained, m.total_marks
        FROM marks m WHERE m.student_id = %s AND m.total_marks IS NOT NULL AND m.marks_obtained IS NOT NULL
        ORDER BY m.exam_date
        """, (student_id,)
    ) or []
    if not all_student:
        return df, None
    s = pd.DataFrame(all_student)
    s["pct"] = (s["marks_obtained"].astype(float)/s["total_marks"].astype(float))*100.0
    student_overall = s["pct"].tail(5).mean() if len(s) >= 1 else s["pct"].mean()
    return df, float(student_overall) if not math.isnan(student_overall) else None

def predict_next_percentage(model_bundle, student_id, subject_id):
    if not model_bundle:
        return None
    model = model_bundle["model"]
    df, student_overall = _latest_stats(student_id, subject_id)
    if df is None or df.empty:
        return None
    df = df.sort_values("exam_date")
    subj_avg = df["pct"].tail(3).mean() if len(df) >= 1 else df["pct"].mean()
    attempts = len(df)
    last_date = df["exam_date"].iloc[-1]
    days_since = max((pd.Timestamp.today() - last_date).days, 1) if pd.notnull(last_date) else 60
    teacher_id = int(df["teacher_id"].iloc[-1]) if "teacher_id" in df.columns else 0
    student_overall = student_overall if student_overall is not None else subj_avg

    X_row = pd.DataFrame([{
        "student_overall_avg": float(student_overall),
        "subj_avg": float(subj_avg),
        "attempts_subj": int(attempts),
        "days_since": float(days_since),
        "subject_id": int(subject_id),
        "teacher_id": int(teacher_id),
    }])
    pred = float(model.predict(X_row)[0])
    return max(0.0, min(100.0, pred))

def percentage_to_grade(pct: float) -> str:
    if pct >= 90: return "A+"
    if pct >= 80: return "A"
    if pct >= 70: return "B"
    if pct >= 60: return "C"
    if pct >= 50: return "D"
    return "F"
```

# Performance Dashboard (performance_dashboard.py)

```
#!/usr/bin/env python3
"""
Student Performance Dashboard - Modern UI matching the design image
"""

import tkinter as tk
from tkinter import ttk, messagebox
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.patches as patches
from matplotlib.patches import Wedge
import numpy as np
from database import db

class PerformanceDashboard:
    def __init__(self, user_type="teacher", user_profile=None):
        self.user_type = user_type
        self.user_profile = user_profile
        self.root = tk.Tk()
        self.root.title("Student Performance Dashboard")
        self.root.geometry("1400x900")
        self.root.state('zoomed')  # Maximize window

        # Configure style
        self.setup_styles()

        # Create main container
        self.main_container = ttk.Frame(self.root)
        self.main_container.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)

        # Create header
        self.create_header()

        # Create filters section
        self.create_filters()

        # Create metrics card
        self.create_metrics_card()

        # Create main content area
        self.create_main_content()

        # Load initial data
        self.load_dashboard_data()

    def setup_styles(self):
        """Setup custom styles for the dashboard"""
        style = ttk.Style()

        # Configure colors to match the design
        style.configure('Header.TLabel', font=('Arial', 24, 'bold'), foreground='#2c3e50')
        style.configure('Card.TFrame', relief='solid', borderwidth=1)
        style.configure('Metric.TLabel', font=('Arial', 14, 'bold'), foreground='#34495e')
        style.configure('Filter.TLabel', font=('Arial', 12), foreground='#7f8c8d')

    def create_header(self):
        """Create header with title and action buttons"""
        header_frame = ttk.Frame(self.main_container)
        header_frame.pack(fill=tk.X, pady=(0, 20))

        # Title
        title_label = ttk.Label(header_frame, text="Student Performance Dashboard",
                                style='Header.TLabel')
        title_label.pack(side=tk.LEFT)

        # Action buttons (right side)
        actions_frame = ttk.Frame(header_frame)
        actions_frame.pack(side=tk.RIGHT)

        # Add refresh, link, expand, and menu buttons (as placeholders)
        ttk.Button(actions_frame, text="■", width=3).pack(side=tk.LEFT, padx=2)
        ttk.Button(actions_frame, text="■", width=3).pack(side=tk.LEFT, padx=2)
        ttk.Button(actions_frame, text="■", width=3).pack(side=tk.LEFT, padx=2)
        ttk.Button(actions_frame, text="■", width=3).pack(side=tk.LEFT, padx=2)

        # User info and logout
        if self.user_profile:
            user_frame = ttk.Frame(header_frame)
            user_frame.pack(side=tk.RIGHT, padx=(20, 0))

            user_label = ttk.Label(user_frame, text=f"Welcome, {self.user_profile.get('fullname', 'User')}",
                                   font=("Arial", 12))
            user_label.pack(side=tk.LEFT, padx=(0, 10))
```

```
        logout_button = ttk.Button(user_frame, text="Logout", command=self.logout)
        logout_button.pack(side=tk.RIGHT)

    def create_filters(self):
        """Create filter section with year and grade dropdowns"""
        filters_frame = ttk.Frame(self.main_container)
        filters_frame.pack(fill=tk.X, pady=(0, 20))

        # Year filter
        year_frame = ttk.Frame(filters_frame)
        year_frame.pack(side=tk.LEFT, padx=(0, 20))

        ttk.Label(year_frame, text="Select Year", style='Filter.TLabel').pack(anchor=tk.W)
        self.year_var = tk.StringVar(value="All")
        year_combo = ttk.Combobox(year_frame, textvariable=self.year_var,
                                  values=["All", "2024", "2023", "2022"],
                                  state="readonly", width=15)
        year_combo.pack(anchor=tk.W, pady=(5, 0))

        # Grade filter
        grade_frame = ttk.Frame(filters_frame)
        grade_frame.pack(side=tk.LEFT)

        ttk.Label(grade_frame, text="Select Grade", style='Filter.TLabel').pack(anchor=tk.W)
        self.grade_var = tk.StringVar(value="All")
        grade_combo = ttk.Combobox(grade_frame, textvariable=self.grade_var,
                                   values=["All", "Grade 1", "Grade 2", "Grade 3", "Grade 4", "Grade 5"],
                                   state="readonly", width=15)
        grade_combo.pack(anchor=tk.W, pady=(5, 0))

        # Bind filter changes
        year_combo.bind('<<ComboboxSelected>>', self.on_filter_change)
        grade_combo.bind('<<ComboboxSelected>>', self.on_filter_change)

    def create_metrics_card(self):
        """Create key metrics card showing total students"""
        metrics_frame = ttk.Frame(self.main_container)
        metrics_frame.pack(fill=tk.X, pady=(0, 20))

        # Create card frame
        card_frame = ttk.Frame(metrics_frame, style='Card.TFrame')
        card_frame.pack(fill=tk.X, padx=10, pady=10)

        # Student icon and count
        content_frame = ttk.Frame(card_frame)
        content_frame.pack(fill=tk.X, padx=20, pady=15)

        # Student icon (using text as placeholder)
        icon_label = ttk.Label(content_frame, text="■", font=("Arial", 24))
        icon_label.pack(side=tk.LEFT, padx=(0, 15))

        # Student count
        count_frame = ttk.Frame(content_frame)
        count_frame.pack(side=tk.LEFT)

        ttk.Label(count_frame, text="Students", style='Filter.TLabel').pack(anchor=tk.W)
        self.student_count_label = ttk.Label(count_frame, text="300",
                                             style='Metric.TLabel', font=("Arial", 20, "bold"))
        self.student_count_label.pack(anchor=tk.W)

    def create_main_content(self):
        """Create main content area with charts"""
        content_frame = ttk.Frame(self.main_container)
        content_frame.pack(fill=tk.BOTH, expand=True)

        # Left column
        left_frame = ttk.Frame(content_frame)
        left_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=(0, 10))

        # Students by Grade and Gender (Donut Chart)
        self.create_donut_chart(left_frame)

        # Examination Results by Branch (Grouped Bar Chart)
        self.create_exam_results_chart(left_frame)

        # Middle column
        middle_frame = ttk.Frame(content_frame)
        middle_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=10)

        # Student Participation Rate by Branch (Horizontal Bar Chart)
        self.create_participation_chart(middle_frame)

        # Right column
        right_frame = ttk.Frame(content_frame)
        right_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=(10, 0))
        # Average Subject Score (Gauge Charts)
```

```
        self.create_gauge_charts(right_frame)

    def create_donut_chart(self, parent):
        """Create donut chart for students by grade and gender"""
        chart_frame = ttk.LabelFrame(parent, text="Students by Grade and Gender", padding="10")
        chart_frame.pack(fill=tk.BOTH, expand=True, pady=(0, 10))

        # Create matplotlib figure
        fig, ax = plt.subplots(figsize=(8, 6))
        fig.patch.set_facecolor('white')

        # Sample data matching the image
        grades = ['Grade 1', 'Grade 2', 'Grade 3', 'Grade 4', 'Grade 5']
        sizes = [22.67, 20.33, 21.33, 14.67, 21.0]
        colors = ['#FFD700', '#FF8C00', '#FF4500', '#FF6347', '#DC143C']

        # Create donut chart
        wedges, texts, autotexts = ax.pie(sizes, labels=grades, colors=colors, autopct='%1.1f%%',
                                          startangle=90, pctdistance=0.85)

        # Create donut hole
        centre_circle = plt.Circle((0,0), 0.70, fc='white')
        ax.add_artist(centre_circle)

        # Style the chart
        ax.set_title('Drill down to show the number of students by gender.',
                     fontsize=10, pad=20, style='italic')

        # Embed in tkinter
        canvas = FigureCanvasTkAgg(fig, chart_frame)
        canvas.draw()
        canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def create_exam_results_chart(self, parent):
        """Create grouped bar chart for examination results"""
        chart_frame = ttk.LabelFrame(parent, text="Examination Results by Branch", padding="10")
        chart_frame.pack(fill=tk.BOTH, expand=True)

        # Create matplotlib figure
        fig, ax = plt.subplots(figsize=(8, 6))
        fig.patch.set_facecolor('white')

        # Sample data matching the image
        subjects = ['Phys. Ed', 'Arts', 'English', 'Science', 'Maths']
        pass_scores = [255, 205, 200, 195, 180]
        fail_scores = [20, 70, 75, 70, 90]
        not_attended = [15, 15, 15, 15, 15]

        x = np.arange(len(subjects))
        width = 0.25

        # Create bars
        bars1 = ax.bar(x - width, pass_scores, width, label='Pass', color='#FFD700')
        bars2 = ax.bar(x, fail_scores, width, label='Fail', color='#DC143C')
        bars3 = ax.bar(x + width, not_attended, width, label='Not attended', color='#8B4513')

        # Style the chart
        ax.set_xlabel('Subjects')
        ax.set_ylabel('Count')
        ax.set_title('Examination Results by Branch')
        ax.set_xticks(x)
        ax.set_xticklabels(subjects)
        ax.legend()
        ax.set_ylim(0, 280)

        plt.tight_layout()

        # Embed in tkinter
        canvas = FigureCanvasTkAgg(fig, chart_frame)
        canvas.draw()
        canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def create_participation_chart(self, parent):
        """Create horizontal bar chart for participation rates"""
        chart_frame = ttk.LabelFrame(parent, text="Student Participation Rate by Branch", padding="10")
        chart_frame.pack(fill=tk.BOTH, expand=True)

        # Create matplotlib figure
        fig, ax = plt.subplots(figsize=(6, 8))
        fig.patch.set_facecolor('white')

        # Sample data matching the image
        subjects = ['English', 'Arts', 'Maths', 'Phy. Ed', 'Science']
        participation = [89, 87.67, 87.33, 85.33, 82.33]
        colors = ['#FFD700'] * len(subjects)
        # Create horizontal bar chart
```

```
        bars = ax.barh(subjects, participation, color=colors)

        # Style the chart
        ax.set_xlabel('Participation Rate (%)')
        ax.set_title('Student Participation Rate by Branch')
        ax.set_xlim(0, 100)

        # Add value labels on bars
        for i, (bar, value) in enumerate(zip(bars, participation)):
            width = bar.get_width()
            ax.text(width + 1, bar.get_y() + bar.get_height()/2,
                    f'{value}%', ha='left', va='center')

        plt.tight_layout()

        # Embed in tkinter
        canvas = FigureCanvasTkAgg(fig, chart_frame)
        canvas.draw()
        canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def create_gauge_charts(self, parent):
        """Create gauge charts for average subject scores"""
        chart_frame = ttk.LabelFrame(parent, text="Avg. Subject Score", padding="10")
        chart_frame.pack(fill=tk.BOTH, expand=True)

        # Create matplotlib figure with subplots
        fig, axes = plt.subplots(2, 3, figsize=(8, 6))
        fig.patch.set_facecolor('white')

        # Sample data matching the image
        subjects = ['Arts', 'English', 'Maths', 'Phy. Ed', 'Science']
        scores = [84.37, 84.05, 81.86, 84.76, 79.36]
        colors = ['#FFD700', '#DC143C', '#FF4500', '#8B4513', '#FF8C00']

        # Flatten axes for easier iteration
        axes_flat = axes.flatten()

        for i, (subject, score, color) in enumerate(zip(subjects, scores, colors)):
            ax = axes_flat[i]

            # Create gauge chart
            self.create_gauge(ax, score, subject, color)

        # Hide the last subplot
        axes_flat[5].set_visible(False)

        plt.tight_layout()

        # Embed in tkinter
        canvas = FigureCanvasTkAgg(fig, chart_frame)
        canvas.draw()
        canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

    def create_gauge(self, ax, value, title, color):
        """Create a single gauge chart"""
        # Calculate angles
        theta = np.linspace(0, np.pi, 100)
        radius = 1

        # Background circle
        ax.plot(radius * np.cos(theta), radius * np.sin(theta), 'k-', linewidth=2)

        # Filled portion
        filled_theta = np.linspace(0, np.pi * (value / 100), 100)
        ax.fill_between(radius * np.cos(filled_theta), 0, radius * np.sin(filled_theta),
                        color=color, alpha=0.7)

        # Add score text
        ax.text(0, 0, f'{value:.2f}', ha='center', va='center', fontsize=10, fontweight='bold')

        # Add title
        ax.text(0, -1.3, title, ha='center', va='center', fontsize=8)

        # Set equal aspect ratio and remove axes
        ax.set_aspect('equal')
        ax.set_xlim(-1.5, 1.5)
        ax.set_ylim(-1.5, 1.5)
        ax.axis('off')

    def load_dashboard_data(self):
        """Load dashboard data from database"""
        try:
            # Get system statistics
            stats = db.get_system_stats()

            # Update student count
```

```
            if stats and 'active_students' in stats:
                try:
                    self.student_count_label.config(text=str(int(stats['active_students'])))
                except Exception:
                    self.student_count_label.config(text=str(stats['active_students']))

            # In a real implementation, you would:
            # 1. Filter data based on year and grade selections
            # 2. Update all charts with real data
            # 3. Handle data refresh when filters change

        except Exception as e:
            print(f"Error loading dashboard data: {e}")

    def on_filter_change(self, event=None):
        """Handle filter changes"""
        # In a real implementation, this would refresh the dashboard data
        # based on the selected year and grade filters
        print(f"Filters changed - Year: {self.year_var.get()}, Grade: {self.grade_var.get()}")
        self.load_dashboard_data()

    def logout(self):
        """Logout and return to login"""
        if messagebox.askyesno("Logout", "Are you sure you want to logout?"):
            self.root.destroy()
            import login
            login.LoginWindow().run()

    def run(self):
        """Run the dashboard"""
        self.root.mainloop()

# Demo function to show the dashboard
def demo_dashboard():
    """Demo function to show the dashboard without login"""
    dashboard = PerformanceDashboard()
    dashboard.run()

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

# Main Entry (main.py)

```
#!/usr/bin/env python3
"""
Main entry point for Student Performance Monitoring System
"""

import sys
import os
from database import db
import admin

def main():
    """Main application function"""
    print("Starting Student Performance Monitoring System...")

    # Check database connection
    if not db.connection or not db.connection.is_connected():
        print("[ERROR] Database connection failed. Please ensure:")
        print("   1. XAMPP is running")
        print("   2. MySQL service is started")
        print("   3. Database 'student_performance_db' exists")
        print("   4. Run database_setup.sql in phpMyAdmin")
        return

    print("[OK] Database connection successful")
    print("Launching Admin Dashboard...")

    # Start the application directly in Admin Dashboard
    try:
        # Try to fetch an existing admin user from DB; fallback to a stub
        admin_user_result = db.execute_query("SELECT user_id, username, role FROM users WHERE role = 'admin'
        if admin_user_result and len(admin_user_result) > 0:
            admin_user = admin_user_result[0]
        else:
            admin_user = {"user_id": 0, "username": "admin", "role": "admin"}

        dashboard = admin.AdminDashboard(admin_user)
        dashboard.run()
    except Exception as e:
        print(f"[ERROR] Application error: {e}")
    finally:
        # Close database connection
        db.close()

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

# Student Dashboard (student.py)

```
#!/usr/bin/env python3
"""
Student Dashboard for Student Performance Monitoring System
"""

import tkinter as tk
from tkinter import ttk, messagebox
from database import db

# Try to import matplotlib, but make it optional
try:
    import matplotlib.pyplot as plt
    from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
    import matplotlib
    matplotlib.use('TkAgg')
    MATPLOTLIB_AVAILABLE = True
except ImportError:
    MATPLOTLIB_AVAILABLE = False
    print("■■ Matplotlib not available. Charts will be disabled.")

class StudentDashboard:
    def __init__(self, user, student_profile):
        self.user = user
        self.student_profile = student_profile
        self.root = tk.Tk()
        self.root.title("Student Performance Dashboard")
        self.root.geometry("1400x900")
        self.root.state('zoomed')  # Maximize window

        # Set modern background color
        self.root.configure(bg='#f5f5f5')

        # Configure grid weights for root window
        self.root.grid_columnconfigure(0, weight=1)
        self.root.grid_rowconfigure(0, weight=1)

        # Configure style
        self.setup_styles()

        # Create scrollable area (prevents content clipping and enables scrolling)
        self._canvas = tk.Canvas(self.root, bg='#f5f5f5', highlightthickness=0)
        self._vsb = ttk.Scrollbar(self.root, orient='vertical', command=self._canvas.yview)
        self._canvas.configure(yscrollcommand=self._vsb.set)
        self._canvas.grid(row=0, column=0, sticky="nsew")
        self._vsb.grid(row=0, column=1, sticky="ns")

        # Inner scrollable frame
        self._scroll_frame = ttk.Frame(self._canvas, style='Main.TFrame')
        self._canvas.create_window((0, 0), window=self._scroll_frame, anchor='nw')
        self._scroll_frame.bind(
            '<Configure>',
            lambda e: self._canvas.configure(scrollregion=self._canvas.bbox('all'))
        )
        # Smooth mouse wheel scrolling (Windows)
        self._canvas.bind_all('<MouseWheel>', lambda e: self._canvas.yview_scroll(int(-1 * (e.delta / 120)),

        # Create main container with modern styling inside scrollable frame
        self.main_container = ttk.Frame(self._scroll_frame, style='Main.TFrame')
        self.main_container.grid(row=0, column=0, sticky="nsew", padx=15, pady=15)

        # Configure grid weights for main container
        self.main_container.grid_columnconfigure(0, weight=1)
        self.main_container.grid_rowconfigure(0, weight=0)  # Header row
        self.main_container.grid_rowconfigure(1, weight=0)  # Filters row
        self.main_container.grid_rowconfigure(2, weight=1)  # Main content row

        # Create header
        self.create_header()

        # Create filters section
        self.create_filters()

        # Create main content area
        self.create_main_content()

        # Load initial data
        self.load_dashboard_data()

    def setup_styles(self):
        """Setup modern styles for the dashboard"""
        style = ttk.Style()

        # Configure modern color scheme
```

```
        style.configure('Main.TFrame', background='#f5f5f5')
        style.configure('Header.TFrame', background='#ffffff', relief='solid', borderwidth=1)
        style.configure('Card.TFrame', background='#ffffff', relief='solid', borderwidth=1)
        style.configure('Header.TLabel', font=('Arial', 24, 'bold'), foreground='#2c3e50', background='#ffff
        style.configure('Metric.TLabel', font=('Arial', 14, 'bold'), foreground='#34495e', background='#ffff
        style.configure('Filter.TLabel', font=('Arial', 12), foreground='#7f8c8d', background='#ffffff')
        style.configure('Stats.TLabel', font=('Arial', 10), foreground='#34495e', background='#ffffff')
        style.configure('StatsValue.TLabel', font=('Arial', 16, 'bold'), foreground='#2c3e50', background='#

    def create_header(self):
        """Create modern header with title and user info using grid layout"""
        header_frame = ttk.Frame(self.main_container, style='Header.TFrame')
        header_frame.grid(row=0, column=0, sticky="ew", pady=(0, 10))
        header_frame.grid_columnconfigure(0, weight=1)
        header_frame.grid_columnconfigure(1, weight=0)

        # Title (left side)
        title_label = ttk.Label(header_frame, text="Student Performance Dashboard",
                                style='Header.TLabel')
        title_label.grid(row=0, column=0, sticky="w", padx=20, pady=15)

        # User info and logout (right side)
        user_frame = ttk.Frame(header_frame, style='Header.TFrame')
        user_frame.grid(row=0, column=1, sticky="e", padx=20, pady=15)
        user_frame.grid_columnconfigure(0, weight=0)
        user_frame.grid_columnconfigure(1, weight=0)
        user_frame.grid_columnconfigure(2, weight=0)

        user_label = ttk.Label(user_frame, text=f"Welcome, {self.student_profile['fullname']}",
                               font=("Arial", 12), background='#ffffff')
        user_label.grid(row=0, column=0, padx=(0, 15))

        # CGPA display in header
        self.cgpa_header_label = ttk.Label(user_frame, text="CGPA: --", font=("Arial", 12, 'bold'), backgrou
        self.cgpa_header_label.grid(row=0, column=1, padx=(0, 15))

        logout_button = ttk.Button(user_frame, text="Logout", command=self.logout)
        logout_button.grid(row=0, column=2)

    def create_filters(self):
        """Create modern filter section with year and grade dropdowns using grid layout"""
        filters_frame = ttk.Frame(self.main_container, style='Card.TFrame')
        filters_frame.grid(row=1, column=0, sticky="ew", pady=(0, 10))
        filters_frame.grid_columnconfigure(0, weight=0)
        filters_frame.grid_columnconfigure(1, weight=0)
        filters_frame.grid_columnconfigure(2, weight=1)

        # Year filter
        year_frame = ttk.Frame(filters_frame, style='Card.TFrame')
        year_frame.grid(row=0, column=0, sticky="w", padx=20, pady=15)

        ttk.Label(year_frame, text="Select Year", style='Filter.TLabel').grid(row=0, column=0, sticky="w")
        self.year_var = tk.StringVar(value="All")
        year_combo = ttk.Combobox(year_frame, textvariable=self.year_var,
                                  values=["All", "2024", "2023", "2022"],
                                  state="readonly", width=15)
        year_combo.grid(row=1, column=0, sticky="w", pady=(5, 0))

        # Grade filter
        grade_frame = ttk.Frame(filters_frame, style='Card.TFrame')
        grade_frame.grid(row=0, column=1, sticky="w", padx=(0, 20), pady=15)

        ttk.Label(grade_frame, text="Select Grade", style='Filter.TLabel').grid(row=0, column=0, sticky="w")
        self.grade_var = tk.StringVar(value="All")
        grade_combo = ttk.Combobox(grade_frame, textvariable=self.grade_var,
                                   values=["All", "Grade 1", "Grade 2", "Grade 3", "Grade 4", "Grade 5"],
                                   state="readonly", width=15)
        grade_combo.grid(row=1, column=0, sticky="w", pady=(5, 0))

        # Bind filter changes
        year_combo.bind('<<ComboboxSelected>>', self.on_filter_change)
        grade_combo.bind('<<ComboboxSelected>>', self.on_filter_change)


    def create_main_content(self):
        """Create modern main content area with 2 columns and full-width sections using grid layout"""
        # Configure grid weights for main content
        self.main_container.grid_rowconfigure(2, weight=1)

        # Main content container
        content_frame = ttk.Frame(self.main_container, style='Main.TFrame')
        content_frame.grid(row=2, column=0, sticky="nsew")
        content_frame.grid_columnconfigure(0, weight=1)
        content_frame.grid_columnconfigure(1, weight=1)
        content_frame.grid_rowconfigure(0, weight=1)  # Charts row
        content_frame.grid_rowconfigure(1, weight=1)  # Charts row 2
```

```
        content_frame.grid_rowconfigure(2, weight=0)  # Recent grades row
        content_frame.grid_rowconfigure(3, weight=0)  # Statistics row

        # Top row - 2 charts side by side (balanced columns)
        charts_row = ttk.Frame(content_frame, style='Main.TFrame')
        charts_row.grid(row=0, column=0, columnspan=2, sticky="nsew", pady=(0, 10))
        charts_row.grid_columnconfigure(0, weight=1)
        charts_row.grid_columnconfigure(1, weight=1)
        charts_row.grid_rowconfigure(0, weight=1)

        # Left chart - Performance Over Time
        left_chart_frame = ttk.Frame(charts_row, style='Card.TFrame')
        left_chart_frame.grid(row=0, column=0, sticky="nsew", padx=(0, 5))
        self.create_performance_chart(left_chart_frame)

        # Right chart - Subject Performance
        right_chart_frame = ttk.Frame(charts_row, style='Card.TFrame')
        right_chart_frame.grid(row=0, column=1, sticky="nsew", padx=(5, 0))
        self.create_subject_chart(right_chart_frame)

        # Second row - additional charts (pie and boxplot)
        charts_row2 = ttk.Frame(content_frame, style='Main.TFrame')
        charts_row2.grid(row=1, column=0, columnspan=2, sticky="nsew", pady=(0, 10))
        charts_row2.grid_columnconfigure(0, weight=1)
        charts_row2.grid_columnconfigure(1, weight=1)
        charts_row2.grid_rowconfigure(0, weight=1)

        # Left chart - Grade Distribution Pie
        pie_frame = ttk.Frame(charts_row2, style='Card.TFrame')
        pie_frame.grid(row=0, column=0, sticky="nsew", padx=(0, 5))
        self.create_grade_distribution_pie(pie_frame)

        # Right chart - Subject Score Boxplot
        box_frame = ttk.Frame(charts_row2, style='Card.TFrame')
        box_frame.grid(row=0, column=1, sticky="nsew", padx=(5, 0))
        self.create_subject_boxplot(box_frame)

        # Recent Grades section (full width)
        grades_frame = ttk.Frame(content_frame, style='Card.TFrame')
        grades_frame.grid(row=2, column=0, columnspan=2, sticky="ew", pady=(0, 10))
        grades_frame.grid_columnconfigure(0, weight=1)
        grades_frame.grid_rowconfigure(0, weight=1)
        self.create_grades_summary(grades_frame)

        # Performance Statistics section (full width)
        stats_frame = ttk.Frame(content_frame, style='Card.TFrame')
        stats_frame.grid(row=3, column=0, columnspan=2, sticky="ew")
        stats_frame.grid_columnconfigure(0, weight=1)
        stats_frame.grid_rowconfigure(0, weight=1)
        self.create_performance_stats(stats_frame)

    def create_grade_distribution_pie(self, parent):
        """Create grade distribution pie chart using real marks"""
        try:
            import matplotlib.pyplot as plt
            from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
            import numpy as np

            parent.grid_columnconfigure(0, weight=1)
            parent.grid_rowconfigure(0, weight=1)

            chart_frame = ttk.LabelFrame(parent, text="My Grade Distribution", padding="10")
            chart_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
            chart_frame.grid_columnconfigure(0, weight=1)
            chart_frame.grid_rowconfigure(0, weight=1)

            fig, ax = plt.subplots(figsize=(6.5, 3.8))
            fig.patch.set_facecolor('white')

            marks = db.get_student_marks(self.student_profile['student_id'])

            if marks:
                percentages = [
                    (float(m['marks_obtained']) / float(m['total_marks'])) * 100 if m['total_marks'] else 0.
                    for m in marks
                ]
                grades = [self.calculate_grade(p) for p in percentages]
                order = ['A+', 'A', 'B', 'C', 'D', 'F']
                counts = [grades.count(g) for g in order]
                labels = [f"{g} ({c})" for g, c in zip(order, counts)]
                colors = ['#2ecc71', '#27ae60', '#3498db', '#f1c40f', '#e67e22', '#e74c3c']

                # Avoid all-zero data
                if sum(counts) > 0:
                    wedges, texts, autotexts = ax.pie(counts, labels=labels, autopct='%1.0f%%',
                                                      startangle=90, colors=colors, pctdistance=0.8)
```

```
                centre_circle = plt.Circle((0,0), 0.55, fc='white')
                ax.add_artist(centre_circle)
                ax.set_title('Grade Distribution', fontsize=12, pad=10)
            else:
                ax.text(0.5, 0.5, 'No data', ha='center', va='center', transform=ax.transAxes)
        else:
            ax.text(0.5, 0.5, 'No data available', ha='center', va='center', transform=ax.transAxes)

        canvas = FigureCanvasTkAgg(fig, chart_frame)
        canvas.draw()
        canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

    except ImportError:
        ttk.Label(parent, text="Matplotlib not available for charts",
                  font=("Arial", 12)).grid(row=0, column=0, sticky="nsew")

def create_subject_boxplot(self, parent):
    """Create subject-wise bar chart (attempts per subject)"""
    try:
        import matplotlib.pyplot as plt
        from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
        import numpy as np

        parent.grid_columnconfigure(0, weight=1)
        parent.grid_rowconfigure(0, weight=1)

        chart_frame = ttk.LabelFrame(parent, text="Attempts per Subject (Bar)", padding="10")
        chart_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
        chart_frame.grid_columnconfigure(0, weight=1)
        chart_frame.grid_rowconfigure(0, weight=1)

        fig, ax = plt.subplots(figsize=(6.5, 3.8))
        fig.patch.set_facecolor('white')

        marks = db.get_student_marks(self.student_profile['student_id'])

        if marks:
            # Count attempts per subject
            counts = {}
            for m in marks:
                subj = str(m['subject_name'])
                counts[subj] = counts.get(subj, 0) + 1

            items = sorted(counts.items(), key=lambda kv: kv[0])[:10]
            labels = [k for k, _ in items]
            values = [v for _, v in items]

            if values:
                colors = ['#2E86AB', '#27ae60', '#f39c12', '#e74c3c', '#9b59b6']
                xs = np.arange(len(labels))
                bars = ax.bar(xs, values, color=[colors[i % len(colors)] for i in range(len(labels))])
                ax.set_ylabel('Attempts')
                ax.set_title('Attempts per Subject', fontsize=12)
                ax.set_xticks(xs)
                ax.set_xticklabels(labels, rotation=20, ha='right')
                for b, v in zip(bars, values):
                    ax.text(b.get_x() + b.get_width()/2, b.get_height()+0.05, f"{int(v)}", ha='center', 
                fig.subplots_adjust(bottom=0.25, left=0.07, right=0.98, top=0.90)
            else:
                ax.text(0.5, 0.5, 'No data', ha='center', va='center', transform=ax.transAxes)
        else:
            ax.text(0.5, 0.5, 'No data available', ha='center', va='center', transform=ax.transAxes)

        canvas = FigureCanvasTkAgg(fig, chart_frame)
        canvas.draw()
        canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

    except ImportError:
        ttk.Label(parent, text="Matplotlib not available for charts",
                  font=("Arial", 12)).grid(row=0, column=0, sticky="nsew")

def create_performance_chart(self, parent):
    """Create performance over time chart with modern styling"""
    try:
        import matplotlib.pyplot as plt
        from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
        import numpy as np

        # Configure parent frame
        parent.grid_columnconfigure(0, weight=1)
        parent.grid_rowconfigure(0, weight=1)

        chart_frame = ttk.LabelFrame(parent, text="My Performance Over Time", padding="10")
        chart_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
        chart_frame.grid_columnconfigure(0, weight=1)
        chart_frame.grid_rowconfigure(0, weight=1)
```

```
            fig, ax = plt.subplots(figsize=(7.0, 3.8))
            fig.patch.set_facecolor('white')

            # Get student's marks
            marks = db.get_student_marks(self.student_profile['student_id'])

            if marks:
                # Prepare data
                # Normalize dates to short strings
                def _fmt_date(d):
                    return d.strftime('%Y-%m-%d') if hasattr(d, 'strftime') else str(d)
                dates = [_fmt_date(mark['exam_date']) for mark in marks]
                percentages = [
                    (float(mark['marks_obtained']) / float(mark['total_marks'])) * 100 if mark['total_marks'
                    for mark in marks
                ]
                subjects = [mark['subject_name'] for mark in marks]

                # Create line plot
                ax.plot(range(len(dates)), percentages, marker='o', linewidth=3, markersize=8, color='#2E86A
                ax.set_title('My Performance Over Time', fontsize=14, fontweight='bold')
                ax.set_ylabel('Percentage (%)', fontsize=12)
                ax.set_xlabel('Exams', fontsize=12)
                ax.grid(True, alpha=0.3)

                # Set x-axis labels
                ax.set_xticks(range(len(dates)))
                # Limit labels to avoid overlap and use tighter rotation
                labels = [f"{s}\n{d}" for s, d in zip(subjects, dates)]
                ax.set_xticklabels(labels, rotation=25, ha='right', fontsize=9)
                ax.set_ylim(0, 100)
                ax.margins(x=0.02, y=0.1)
                fig.subplots_adjust(bottom=0.32, left=0.08, right=0.98, top=0.90)

                # Add average line
                avg_percentage = sum(percentages) / len(percentages)
                ax.axhline(y=avg_percentage, color='red', linestyle='--', alpha=0.7, label=f'Average: {avg_p
                ax.legend()

                plt.tight_layout()
            else:
                ax.text(0.5, 0.5, 'No performance data available',
                        ha='center', va='center', transform=ax.transAxes, fontsize=12)
                ax.set_title('My Performance Over Time', fontsize=14, fontweight='bold')

            canvas = FigureCanvasTkAgg(fig, chart_frame)
            canvas.draw()
            canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

        except ImportError:
            ttk.Label(parent, text="Matplotlib not available for charts",
                      font=("Arial", 12)).grid(row=0, column=0, sticky="nsew")

    def create_subject_chart(self, parent):
        """Create subject performance chart with modern styling"""
        try:
            import matplotlib.pyplot as plt
            from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
            import numpy as np

            # Configure parent frame
            parent.grid_columnconfigure(0, weight=1)
            parent.grid_rowconfigure(0, weight=1)

            chart_frame = ttk.LabelFrame(parent, text="My Average Performance by Subject", padding="10")
            chart_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
            chart_frame.grid_columnconfigure(0, weight=1)
            chart_frame.grid_rowconfigure(0, weight=1)

            fig, ax = plt.subplots(figsize=(7.0, 4.1))
            fig.patch.set_facecolor('white')

            # Get student's marks
            marks = db.get_student_marks(self.student_profile['student_id'])

            if marks:
                # Group by subject and compute averages
                subject_marks = {}
                for mark in marks:
                    subject = str(mark['subject_name'])
                    percentage = (
                        (float(mark['marks_obtained']) / float(mark['total_marks'])) * 100
                        if mark['total_marks'] else 0.0
                    )
                    subject_marks.setdefault(subject, []).append(percentage)
                items = [
```

```
                    (subj, sum(vals) / len(vals)) for subj, vals in subject_marks.items()
                ]
                # Sort by average desc and limit to top 10 to reduce clutter
                items.sort(key=lambda x: x[1], reverse=True)
                items = items[:10]
                # Unpack reversed so highest appears at top in barh
                subjects = [s for s, _ in items][::-1]
                avg_percentages = [v for _, v in items][::-1]

                # Create horizontal bar chart
                colors = ['#2E86AB', '#27ae60', '#f39c12', '#e74c3c', '#9b59b6']
                bars = ax.barh(subjects, avg_percentages, color=colors * ((len(subjects)//len(colors))+1))
                ax.set_title('My Average Performance by Subject', fontsize=14, fontweight='bold')
                ax.set_xlabel('Average Percentage (%)', fontsize=12)
                ax.set_xlim(0, 100)
                ax.margins(y=0.05)
                fig.subplots_adjust(left=0.32, right=0.95, top=0.92, bottom=0.12)

                # Add value labels on bars
                for i, (bar, value) in enumerate(zip(bars, avg_percentages)):
                    width = bar.get_width()
                    ax.text(min(width + 1, 98), bar.get_y() + bar.get_height()/2,
                           f'{value:.1f}%', ha='left', va='center', fontweight='bold', fontsize=9)

                plt.tight_layout()
            else:
                ax.text(0.5, 0.5, 'No performance data available',
                       ha='center', va='center', transform=ax.transAxes, fontsize=12)
                ax.set_title('My Average Performance by Subject', fontsize=14, fontweight='bold')

            canvas = FigureCanvasTkAgg(fig, chart_frame)
            canvas.draw()
            canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

        except ImportError:
            ttk.Label(parent, text="Matplotlib not available for charts",
                     font=("Arial", 12)).grid(row=0, column=0, sticky="nsew")

    def create_performance_stats(self, parent):
        """Create modern performance statistics with horizontal cards layout"""
        # Configure parent frame
        parent.grid_columnconfigure(0, weight=1)
        parent.grid_rowconfigure(0, weight=1)

        stats_frame = ttk.LabelFrame(parent, text="My Performance Statistics", padding="10")
        stats_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
        stats_frame.grid_columnconfigure(0, weight=1)
        stats_frame.grid_rowconfigure(0, weight=1)

        # Get student's marks
        marks = db.get_student_marks(self.student_profile['student_id'])

        if marks:
            # Calculate comprehensive statistics
            total_marks = len(marks)
            percentages = [
                (float(m['marks_obtained']) / float(m['total_marks'])) * 100 if m['total_marks'] else 0.0
                for m in marks
            ]
            avg_percentage = sum(percentages) / total_marks
            highest_mark = max(percentages)
            gpa = self.calculate_gpa(marks)

            # Create horizontal cards layout
            cards_frame = ttk.Frame(stats_frame, style='Card.TFrame')
            cards_frame.grid(row=0, column=0, sticky="ew")
            cards_frame.grid_columnconfigure(0, weight=1)
            cards_frame.grid_columnconfigure(1, weight=1)
            cards_frame.grid_columnconfigure(2, weight=1)
            cards_frame.grid_columnconfigure(3, weight=1)

            # Create metric cards
            metrics = [
                ("Average", f"{avg_percentage:.1f}%", "#2E86AB"),
                ("CGPA", f"{gpa:.2f}", "#27ae60"),
                ("Best Score", f"{highest_mark:.1f}%", "#f39c12"),
                ("Total Exams", str(total_marks), "#e74c3c")
            ]

            for i, (label, value, color) in enumerate(metrics):
                card = ttk.Frame(cards_frame, style='Card.TFrame')
                card.grid(row=0, column=i, sticky="ew", padx=5)
                card.grid_columnconfigure(0, weight=1)
                card.grid_rowconfigure(0, weight=1)

                # Card content
```

```
                content_frame = ttk.Frame(card, style='Card.TFrame')
                content_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
                content_frame.grid_columnconfigure(0, weight=1)

                # Label
                ttk.Label(content_frame, text=label, style='Stats.TLabel').grid(row=0, column=0, pady=(0, 5)

                # Value
                ttk.Label(content_frame, text=value, style='StatsValue.TLabel', foreground=color).grid(row=1

        else:
            # No data message
            no_data_label = ttk.Label(stats_frame, text="No performance data available",
                                      font=("Arial", 14), background='#ffffff')
            no_data_label.grid(row=0, column=0, sticky="nsew")

    def create_gauge(self, ax, value, title, color, max_val=100):
        """Create a single gauge chart"""
        import numpy as np

        # Calculate angles
        theta = np.linspace(0, np.pi, 100)
        radius = 1

        # Background circle
        ax.plot(radius * np.cos(theta), radius * np.sin(theta), 'k-', linewidth=2)

        # Filled portion
        filled_theta = np.linspace(0, np.pi * (value / max_val), 100)
        ax.fill_between(radius * np.cos(filled_theta), 0, radius * np.sin(filled_theta),
                        color=color, alpha=0.7)

        # Add value text
        ax.text(0, 0, f'{value:.1f}', ha='center', va='center', fontsize=10, fontweight='bold')

        # Add title
        ax.text(0, -1.3, title, ha='center', va='center', fontsize=8)

        # Set equal aspect ratio and remove axes
        ax.set_aspect('equal')
        ax.set_xlim(-1.5, 1.5)
        ax.set_ylim(-1.5, 1.5)
        ax.axis('off')

    def create_grades_summary(self, parent):
        """Create modern grades summary panel with recent grades table"""
        # Configure parent frame
        parent.grid_columnconfigure(0, weight=1)
        parent.grid_rowconfigure(0, weight=1)

        summary_frame = ttk.LabelFrame(parent, text="Recent Grades Performance", padding="10")
        summary_frame.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)
        summary_frame.grid_columnconfigure(0, weight=1)
        summary_frame.grid_rowconfigure(0, weight=1)

        # Get student's marks
        marks = db.get_student_marks(self.student_profile['student_id'])

        if marks:
            # Create modern table for recent grades
            table_frame = ttk.Frame(summary_frame, style='Card.TFrame')
            table_frame.grid(row=0, column=0, sticky="nsew")
            table_frame.grid_columnconfigure(0, weight=1)
            table_frame.grid_rowconfigure(0, weight=1)

            columns = ('Subject', 'Grade', 'Score', 'Date', 'Status')
            self.grades_tree = ttk.Treeview(table_frame, columns=columns, show='headings', height=6)

            # Configure columns
            self.grades_tree.heading('Subject', text='Subject')
            self.grades_tree.heading('Grade', text='Grade')
            self.grades_tree.heading('Score', text='Score')
            self.grades_tree.heading('Date', text='Date')
            self.grades_tree.heading('Status', text='Status')

            self.grades_tree.column('Subject', width=150, anchor='w')
            self.grades_tree.column('Grade', width=80, anchor='center')
            self.grades_tree.column('Score', width=100, anchor='center')
            self.grades_tree.column('Date', width=120, anchor='center')
            self.grades_tree.column('Status', width=100, anchor='center')

            # Add scrollbar
            scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=self.grades_tree.yview)
            self.grades_tree.configure(yscrollcommand=scrollbar.set)

            self.grades_tree.grid(row=0, column=0, sticky="nsew")
```

```
            scrollbar.grid(row=0, column=1, sticky="ns")

            # Populate with recent marks
            recent_marks = marks[:10]  # Show last 10 grades
            for mark in recent_marks:
                percentage = (
                    (float(mark['marks_obtained']) / float(mark['total_marks'])) * 100
                    if mark['total_marks'] else 0.0
                )
                grade = self.calculate_grade(percentage)

                # Determine status based on grade
                if grade in ['A+', 'A']:
                    status = "Excellent"
                elif grade == 'B':
                    status = "Good"
                elif grade == 'C':
                    status = "Average"
                elif grade == 'D':
                    status = "Below Average"
                else:
                    status = "Needs Improvement"

                self.grades_tree.insert('', 'end', values=(
                    mark['subject_name'],
                    grade,
                    f"{percentage:.1f}%",
                    mark['exam_date'],
                    status
                ))
        else:
            # No data message
            no_data_label = ttk.Label(summary_frame, text="No grades data available",
                                      font=("Arial", 14), background='#ffffff')
            no_data_label.grid(row=0, column=0, sticky="nsew")

    def on_filter_change(self, event=None):
        """Handle filter changes"""
        print(f"Filters changed - Year: {self.year_var.get()}, Grade: {self.grade_var.get()}")
        # In a real implementation, this would refresh the dashboard data
        # based on the selected year and grade filters
        self.load_dashboard_data()

    def load_dashboard_data(self):
        """Load dashboard data from database"""
        try:
            # Get student's marks
            marks = db.get_student_marks(self.student_profile['student_id'])

            # Update CGPA in header
            try:
                cgpa = self.calculate_gpa(marks)
                self.cgpa_header_label.config(text=f"CGPA: {cgpa:.2f}")
            except Exception:
                self.cgpa_header_label.config(text="CGPA: --")

            # Update any dynamic content based on filters (placeholder)

        except Exception as e:
            print(f"Error loading dashboard data: {e}")


    def calculate_grade(self, percentage):
        """Calculate letter grade based on percentage"""
        if percentage >= 90:
            return 'A+'
        elif percentage >= 80:
            return 'A'
        elif percentage >= 70:
            return 'B'
        elif percentage >= 60:
            return 'C'
        elif percentage >= 50:
            return 'D'
        else:
            return 'F'

    def calculate_gpa(self, marks):
        """Calculate CGPA as credits-weighted average of subject grade points.
        CGPA = Σ(grade_points(subject_avg) × subject_credits) ÷ Σ(subject_credits)
        Subject average is computed from all attempts/assessments in that subject.
        """
        if not marks:
            return 0.0

        # Group by subject and calculate subject-level average percentage
```

```
        by_subject = {}
        for m in marks:
            subject_id = m.get('subject_id')
            subject_name = str(m.get('subject_name'))
            key = (subject_id, subject_name)
            pct = (
                (float(m.get('marks_obtained', 0)) / float(m.get('total_marks', 1))) * 100
                if m.get('total_marks') else 0.0
            )
            credits = int(m.get('credits') or 3)
            entry = by_subject.setdefault(key, { 'percentages': [], 'credits': credits })
            entry['percentages'].append(pct)
            if credits:
                entry['credits'] = credits

        total_points = 0.0
        total_credits = 0
        for (_, _), info in by_subject.items():
            if not info['percentages']:
                continue
            avg_pct = sum(info['percentages']) / len(info['percentages'])
            gp = self.get_grade_points(avg_pct)
            cr = max(int(info.get('credits') or 0), 0)
            total_points += gp * cr
            total_credits += cr

        return (total_points / total_credits) if total_credits > 0 else 0.0

    def get_grade_points(self, percentage):
        """Get grade points based on percentage"""
        if percentage >= 90:
            return 4.0
        elif percentage >= 80:
            return 3.7
        elif percentage >= 70:
            return 3.0
        elif percentage >= 60:
            return 2.0
        elif percentage >= 50:
            return 1.0
        else:
            return 0.0


    def logout(self):
        """Logout and return to login"""
        if messagebox.askyesno("Logout", "Are you sure you want to logout?"):
            # Clean up bindings
            self.root.unbind_all("<MouseWheel>")
            self.root.unbind_all("<Key>")
            self.root.destroy()
            import login
            login.LoginWindow().run()

    def run(self):
        """Run the dashboard"""
        # Bind cleanup when window is closed
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
        self.root.mainloop()

    def on_closing(self):
        """Handle window closing"""
        # Clean up bindings
        self.root.unbind_all("<MouseWheel>")
        self.root.unbind_all("<Key>")
        self.root.destroy()
```

# Teacher Dashboard (teacher.py)

```
#!/usr/bin/env python3
"""
Teacher Dashboard for Student Performance Monitoring System
"""

import tkinter as tk
from tkinter import ttk, messagebox
from database import db
import datetime
from performance_dashboard import PerformanceDashboard

class TeacherDashboard:
    def __init__(self, user, teacher_profile):
        self.user = user
        self.teacher_profile = teacher_profile
        self.root = tk.Tk()
        self.root.title("Teacher Dashboard - Student Performance Monitoring System")
        self.root.geometry("1400x900")
        self.root.state('zoomed')  # Maximize window

        # Configure grid weights for root window
        self.root.grid_columnconfigure(0, weight=1)
        self.root.grid_rowconfigure(0, weight=1)

        # Create scrollable area for the whole dashboard
        self._canvas = tk.Canvas(self.root, bg=self.root.cget('bg'), highlightthickness=0)
        self._vsb = ttk.Scrollbar(self.root, orient='vertical', command=self._canvas.yview)
        self._canvas.configure(yscrollcommand=self._vsb.set)
        self._canvas.grid(row=0, column=0, sticky="nsew")
        self._vsb.grid(row=0, column=1, sticky="ns")

        # Inner scrollable frame
        self._scroll_frame = ttk.Frame(self._canvas)
        self._canvas.create_window((0, 0), window=self._scroll_frame, anchor='nw')
        self._scroll_frame.bind(
            '<Configure>',
            lambda e: self._canvas.configure(scrollregion=self._canvas.bbox('all'))
        )
        # Smooth mouse wheel scrolling (Windows)
        self._canvas.bind_all('<MouseWheel>', lambda e: self._canvas.yview_scroll(int(-1 * (e.delta / 120)),

        # Main container inside the scrollable frame
        self.main_container = ttk.Frame(self._scroll_frame)
        self.main_container.grid(row=0, column=0, sticky="nsew", padx=10, pady=10)

        # Configure grid weights for main container
        self.main_container.grid_columnconfigure(0, weight=1)
        self.main_container.grid_rowconfigure(0, weight=0)  # Header row
        self.main_container.grid_rowconfigure(1, weight=1)  # Notebook row

        # Create header
        self.create_header()

        # Create notebook for tabs
        self.notebook = ttk.Notebook(self.main_container)
        self.notebook.grid(row=1, column=0, sticky="nsew", pady=(10, 0))

        # Create tabs
        self.create_performance_tab()
        self.create_students_tab()
        self.create_marks_tab()

        # Load initial data
        self.load_dashboard_data()

    def create_header(self):
        """Create header with title and logout button using grid layout"""
        header_frame = ttk.Frame(self.main_container)
        header_frame.grid(row=0, column=0, sticky="ew", pady=(0, 10))
        header_frame.grid_columnconfigure(0, weight=1)
        header_frame.grid_columnconfigure(1, weight=0)

        # Title
        title_label = ttk.Label(header_frame, text="Teacher Dashboard",
                                font=("Arial", 20, "bold"))
        title_label.grid(row=0, column=0, sticky="w")

        # User info and logout
        user_frame = ttk.Frame(header_frame)
        user_frame.grid(row=0, column=1, sticky="e")

        user_label = ttk.Label(user_frame, text=f"Welcome, {self.teacher_profile['fullname']}",
                               font=("Arial", 12))
```

```
        user_label.grid(row=0, column=0, padx=(0, 10))

        logout_button = ttk.Button(user_frame, text="Logout", command=self.logout)
        logout_button.grid(row=0, column=1)

    def create_performance_tab(self):
        """Create performance dashboard tab"""
        self.performance_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.performance_frame, text="Performance Dashboard")

        # Create embedded performance dashboard
        self.performance_dashboard = PerformanceDashboard("teacher", self.teacher_profile)

        # Move the performance dashboard content to this tab
        # We'll create a simplified version that fits in the tab
        self.create_embedded_performance_dashboard()

    def create_embedded_performance_dashboard(self):
        """Create embedded performance dashboard for the tab using grid layout"""
        # Configure grid weights for performance frame
        self.performance_frame.grid_columnconfigure(0, weight=1)
        self.performance_frame.grid_rowconfigure(0, weight=0)  # Header row
        self.performance_frame.grid_rowconfigure(1, weight=0)  # Filters row
        self.performance_frame.grid_rowconfigure(2, weight=0)  # Metrics row
        self.performance_frame.grid_rowconfigure(3, weight=1)  # Charts row

        # Header
        header_frame = ttk.Frame(self.performance_frame)
        header_frame.grid(row=0, column=0, sticky="ew", pady=(0, 20))
        header_frame.grid_columnconfigure(0, weight=1)

        title_label = ttk.Label(header_frame, text="Student Performance Dashboard",
                                font=("Arial", 20, "bold"))
        title_label.grid(row=0, column=0, sticky="w")

        # Filters
        filters_frame = ttk.Frame(self.performance_frame)
        filters_frame.grid(row=1, column=0, sticky="ew", pady=(0, 20))
        filters_frame.grid_columnconfigure(0, weight=0)
        filters_frame.grid_columnconfigure(1, weight=0)
        filters_frame.grid_columnconfigure(2, weight=1)

        # Year filter
        year_frame = ttk.Frame(filters_frame)
        year_frame.grid(row=0, column=0, sticky="w", padx=(0, 20))

        ttk.Label(year_frame, text="Select Year", font=("Arial", 12)).grid(row=0, column=0, sticky="w")
        self.year_var = tk.StringVar(value="All")
        year_combo = ttk.Combobox(year_frame, textvariable=self.year_var,
                                  values=["All", "2024", "2023", "2022"],
                                  state="readonly", width=15)
        year_combo.grid(row=1, column=0, sticky="w", pady=(5, 0))

        # Grade filter
        grade_frame = ttk.Frame(filters_frame)
        grade_frame.grid(row=0, column=1, sticky="w")

        ttk.Label(grade_frame, text="Select Grade", font=("Arial", 12)).grid(row=0, column=0, sticky="w")
        self.grade_var = tk.StringVar(value="All")
        grade_combo = ttk.Combobox(grade_frame, textvariable=self.grade_var,
                                   values=["All", "Grade 1", "Grade 2", "Grade 3", "Grade 4", "Grade 5"],
                                   state="readonly", width=15)
        grade_combo.grid(row=1, column=0, sticky="w", pady=(5, 0))

        # Metrics card
        metrics_frame = ttk.Frame(self.performance_frame)
        metrics_frame.grid(row=2, column=0, sticky="ew", pady=(0, 20))
        metrics_frame.grid_columnconfigure(0, weight=1)

        card_frame = ttk.Frame(metrics_frame, relief='solid', borderwidth=1)
        card_frame.grid(row=0, column=0, sticky="ew", padx=10, pady=10)
        card_frame.grid_columnconfigure(0, weight=1)

        content_frame = ttk.Frame(card_frame)
        content_frame.grid(row=0, column=0, sticky="ew", padx=20, pady=15)
        content_frame.grid_columnconfigure(1, weight=1)

        icon_label = ttk.Label(content_frame, text="■", font=("Arial", 24))
        icon_label.grid(row=0, column=0, sticky="w", padx=(0, 15))

        count_frame = ttk.Frame(content_frame)
        count_frame.grid(row=0, column=1, sticky="w")

        ttk.Label(count_frame, text="Students", font=("Arial", 12)).grid(row=0, column=0, sticky="w")
        self.student_count_label = ttk.Label(count_frame, text="300",
                                             font=("Arial", 20, "bold"))
```

```
        self.student_count_label.grid(row=1, column=0, sticky="w")

        # Charts area
        charts_frame = ttk.Frame(self.performance_frame)
        charts_frame.grid(row=3, column=0, sticky="nsew")
        charts_frame.grid_columnconfigure(0, weight=1)
        charts_frame.grid_columnconfigure(1, weight=1)
        charts_frame.grid_rowconfigure(0, weight=1)

        # Left column - Donut chart
        left_frame = ttk.Frame(charts_frame)
        left_frame.grid(row=0, column=0, sticky="nsew", padx=(0, 10))

        self.create_donut_chart(left_frame)

        # Right column - Bar chart
        right_frame = ttk.Frame(charts_frame)
        right_frame.grid(row=0, column=1, sticky="nsew", padx=(10, 0))

        self.create_bar_chart(right_frame)

    def create_donut_chart(self, parent):
        """Create donut chart for students (gender distribution) using DB"""
        try:
            import matplotlib.pyplot as plt
            from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
            import numpy as np

            chart_frame = ttk.LabelFrame(parent, text="Students by Gender", padding="10")
            chart_frame.grid(row=0, column=0, sticky="nsew")
            chart_frame.grid_columnconfigure(0, weight=1)
            chart_frame.grid_rowconfigure(0, weight=1)

            fig, ax = plt.subplots(figsize=(6, 5))
            fig.patch.set_facecolor('white')

            # Real data (gender distribution of teacher's students)
            gd = db.get_teacher_students_gender_counts(self.teacher_profile['teacher_id'])
            labels = ['Male', 'Female', 'Other']
            sizes = [gd.get('Male', 0), gd.get('Female', 0), gd.get('Other', 0)]
            colors = ['#3498db', '#e74c3c', '#95a5a6']

            wedges, texts, autotexts = ax.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%',
                                              startangle=90, pctdistance=0.85)

            centre_circle = plt.Circle((0,0), 0.70, fc='white')
            ax.add_artist(centre_circle)

            ax.set_title('Gender split of your students',
                         fontsize=10, pad=20, style='italic')

            canvas = FigureCanvasTkAgg(fig, chart_frame)
            canvas.draw()
            canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

        except ImportError:
            ttk.Label(parent, text="Matplotlib not available for charts",
                      font=("Arial", 12)).grid(row=0, column=0, sticky="nsew")

    def create_bar_chart(self, parent):
        """Create bar chart of average percentage by subject using DB"""
        try:
            import matplotlib.pyplot as plt
            from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
            import numpy as np

            chart_frame = ttk.LabelFrame(parent, text="Average Percentage by Subject", padding="10")
            chart_frame.grid(row=0, column=0, sticky="nsew")
            chart_frame.grid_columnconfigure(0, weight=1)
            chart_frame.grid_rowconfigure(0, weight=1)

            fig, ax = plt.subplots(figsize=(6, 5))
            fig.patch.set_facecolor('white')

            # Real data
            rows = db.get_teacher_subject_average_percentages(self.teacher_profile['teacher_id'])
            subjects = [r['subject_name'] for r in rows] or ['No Data']
            averages = [round(float(r.get('avg_pct') or 0), 1) for r in rows] or [0.0]
            x = np.arange(len(subjects))
            bars = ax.bar(x, averages, color='#3498db')
            ax.set_xlabel('Subjects')
            ax.set_ylabel('Average %')
            ax.set_title('Your subjects - average percentage')
            ax.set_xticks(x)
            ax.set_xticklabels(subjects, rotation=0)
            ax.set_ylim(0, 100)
```

```
            plt.tight_layout()

            canvas = FigureCanvasTkAgg(fig, chart_frame)
            canvas.draw()
            canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

        except ImportError:
            ttk.Label(parent, text="Matplotlib not available for charts",
                      font=("Arial", 12)).grid(row=0, column=0, sticky="nsew")

    def create_students_tab(self):
        """Create students tab using grid layout"""
        self.students_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.students_frame, text="My Students")

        # Configure grid weights for students frame
        self.students_frame.grid_columnconfigure(0, weight=1)
        self.students_frame.grid_rowconfigure(0, weight=1)

        # Students table
        table_frame = ttk.Frame(self.students_frame)
        table_frame.grid(row=0, column=0, sticky="nsew", pady=10)
        table_frame.grid_columnconfigure(0, weight=1)
        table_frame.grid_rowconfigure(0, weight=1)

        # Create Treeview
        columns = ('ID', 'Name', 'Email', 'Phone', 'Gender', 'Status')
        self.students_tree = ttk.Treeview(table_frame, columns=columns, show='headings', height=15)

        # Define headings
        for col in columns:
            self.students_tree.heading(col, text=col)
            self.students_tree.column(col, width=120)

        # Add scrollbar
        scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=self.students_tree.yview)
        self.students_tree.configure(yscrollcommand=scrollbar.set)

        self.students_tree.grid(row=0, column=0, sticky="nsew")
        scrollbar.grid(row=0, column=1, sticky="ns")

        # Load sample data
        self.load_students()

    def create_marks_tab(self):
        """Create marks management tab using grid layout"""
        self.marks_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.marks_frame, text="Manage Marks")

        # Configure grid weights for marks frame
        self.marks_frame.grid_columnconfigure(0, weight=1)
        self.marks_frame.grid_rowconfigure(0, weight=0)  # Actions row
        self.marks_frame.grid_rowconfigure(1, weight=1)  # Table row

        # Action buttons
        actions_frame = ttk.Frame(self.marks_frame)
        actions_frame.grid(row=0, column=0, sticky="ew", pady=10)
        actions_frame.grid_columnconfigure(0, weight=1)
        actions_frame.grid_columnconfigure(1, weight=0)
        actions_frame.grid_columnconfigure(2, weight=0)
        actions_frame.grid_columnconfigure(3, weight=0)

        ttk.Button(actions_frame, text="Add Mark",
                   command=self.add_mark).grid(row=0, column=1, padx=5)
        ttk.Button(actions_frame, text="Edit Mark",
                   command=self.edit_mark).grid(row=0, column=2, padx=5)
        ttk.Button(actions_frame, text="Delete Mark",
                   command=self.delete_mark).grid(row=0, column=3, padx=5)

        # Marks table
        table_frame = ttk.Frame(self.marks_frame)
        table_frame.grid(row=1, column=0, sticky="nsew", pady=10)
        table_frame.grid_columnconfigure(0, weight=1)
        table_frame.grid_rowconfigure(0, weight=1)

        # Create Treeview
        columns = ('Student', 'Subject', 'Exam Type', 'Marks', 'Total', 'Percentage', 'Date')
        self.marks_tree = ttk.Treeview(table_frame, columns=columns, show='headings', height=15)

        # Define headings
        for col in columns:
            self.marks_tree.heading(col, text=col)
            self.marks_tree.column(col, width=100)

        # Add scrollbar
        scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL, command=self.marks_tree.yview)
```

```
        self.marks_tree.configure(yscrollcommand=scrollbar.set)

        self.marks_tree.grid(row=0, column=0, sticky="nsew")
        scrollbar.grid(row=0, column=1, sticky="ns")

        # Load sample data
        self.load_marks()

    def load_dashboard_data(self):
        """Load dashboard data with real DB values for this teacher"""
        # Update Students count metric
        try:
            students = db.get_teacher_students(self.teacher_profile['teacher_id'])
            self.student_count_label.config(text=str(len(students)))
        except Exception:
            self.student_count_label.config(text="0")

    def load_students(self):
        """Load students data"""
        # Clear existing items
        for item in self.students_tree.get_children():
            self.students_tree.delete(item)

        # Load students taught by this teacher (distinct based on marks)
        students = db.get_teacher_students(self.teacher_profile['teacher_id'])
        for student in students:
            self.students_tree.insert('', 'end', values=(
                student['student_id'],
                student['fullname'],
                student['email'],
                student['phone'],
                student['gender'],
                student['status']
            ))

    def load_marks(self):
        """Load marks for this teacher from DB"""
        for item in self.marks_tree.get_children():
            self.marks_tree.delete(item)
        rows = db.get_marks_for_teacher(self.teacher_profile['teacher_id'])
        for r in rows:
            percent = 0
            try:
                if r['total_marks']:
                    percent = round((float(r['marks_obtained']) / float(r['total_marks'])) * 100)
            except Exception:
                percent = 0
            date_str = r['exam_date'].strftime('%Y-%m-%d') if hasattr(r['exam_date'], 'strftime') else (r['e
            self.marks_tree.insert('', 'end', iid=str(r['mark_id']), values=(
                r['student_name'],
                r['subject_name'],
                '',  # exam type not modeled
                r['marks_obtained'],
                r['total_marks'],
                f"{percent}%",
                date_str,
            ))

    def add_mark(self):
        """Add new mark"""
        form = tk.Toplevel(self.root)
        form.title("Add Mark")
        form.geometry("460x380")
        form.resizable(False, False)

        row = 0
        def add_row(label_text, widget):
            ttk.Label(form, text=label_text).grid(row=row, column=0, padx=16, pady=8, sticky="w")
            widget.grid(row=row, column=1, padx=16, pady=8, sticky="w")
            return 1

        # Student selection
        students = db.get_all_students() or []
        student_map = { f"{s['student_id']} - {s['fullname']}": s['student_id'] for s in students }
        student_var = tk.StringVar()
        student_combo = ttk.Combobox(form, textvariable=student_var, values=list(student_map.keys()), width=
        row += add_row("Student", student_combo)

        # Subject selection (for this teacher) - names only, allow typing
        subjects = db.get_teacher_subjects(self.teacher_profile['teacher_id'])
        # Map subject name -> id for resolution on submit
        subject_map = { str(sub['subject_name']): sub['subject_id'] for sub in subjects }
        subject_var = tk.StringVar()
        subject_combo = ttk.Combobox(form, textvariable=subject_var, values=list(subject_map.keys()), width=
        row += add_row("Subject", subject_combo)
        marks_var = tk.StringVar()
```

```
total_var = tk.StringVar()
today_str = datetime.date.today().strftime('%Y-%m-%d')
date_var = tk.StringVar(value=today_str)
marks_entry = ttk.Entry(form, textvariable=marks_var, width=30)
total_entry = ttk.Entry(form, textvariable=total_var, width=30)
row += add_row("Marks Obtained", marks_entry)
# Default total to 100; will auto-adjust if marks exceed total
total_var.set("100")
row += add_row("Total Marks", total_entry)
# Live percentage display
percent_var = tk.StringVar(value="")
percent_label = ttk.Label(form, textvariable=percent_var)
row += add_row("Percentage", percent_label)
# Date row with calendar icon button
date_entry = ttk.Entry(form, textvariable=date_var, width=30)
# Ensure the entry visibly shows today's date if empty
if not (date_entry.get() or "").strip():
    date_entry.insert(0, today_str)
# Place date label, entry, and calendar icon in the same row
tk.Label(form, text="Exam Date (YYYY-MM-DD)").grid(row=row, column=0, padx=16, pady=8, sticky="w")
date_entry.grid(row=row, column=1, padx=16, pady=8, sticky="w")
def set_today():
    date_var.set(datetime.date.today().strftime('%Y-%m-%d'))
    try:
        date_entry.icursor('end')
    except Exception:
        pass
calendar_btn = ttk.Button(form, text="■", width=3, command=set_today)
calendar_btn.grid(row=row, column=2, padx=(0, 0), pady=8, sticky="w")
row += 1

# Helpers to keep total reasonable and update percentage
def _update_percent_and_total(*_):
    m_txt = (marks_var.get() or "").strip()
    t_txt = (total_var.get() or "").strip()
    try:
        m_val = float(m_txt.replace(',', '.')) if m_txt else None
    except Exception:
        m_val = None
    # Default total to 100 if empty
    if not t_txt:
        total_var.set("100")
        t_txt = "100"
    # If marks exceed total, auto-raise total to marks
    try:
        t_val = float(t_txt.replace(',', '.')) if t_txt else None
    except Exception:
        t_val = None
    if m_val is not None and t_val is not None and m_val > t_val:
        total_var.set(str(m_val))
        t_val = m_val
    # Update percentage
    if m_val is not None and t_val not in (None, 0):
        pct = round((m_val / t_val) * 100, 1)
        percent_var.set(f"{pct}%")
    else:
        percent_var.set("")
# Trace variable changes
try:
    marks_var.trace_add('write', _update_percent_and_total)
    total_var.trace_add('write', _update_percent_and_total)
except Exception:
    # Fallback for older Tk versions
    marks_var.trace('w', _update_percent_and_total)
    total_var.trace('w', _update_percent_and_total)
# Initial compute
_update_percent_and_total()

# Preselect first options if available to avoid empty selection
try:
    if student_combo['values']:
        student_combo.current(0)
    if subject_combo['values']:
        subject_combo.current(0)
except Exception:
    pass

def submit():
    # Resolve student selection robustly
    student_key = (student_var.get() or student_combo.get() or "").strip()
    if not student_key and getattr(student_combo, '__getitem__', None):
        try:
            vals = student_combo['values']
            if vals:
                student_key = vals[0]
        except Exception:
```

```
                pass
            try:
                student_id = None
                if ' - ' in student_key:
                    student_id = int(student_key.split(' - ', 1)[0])
                else:
                    # Try mapping fallback or direct int
                    student_id = student_map.get(student_key) or int(student_key)
            except Exception:
                student_id = None
            if not student_id:
                messagebox.showerror("Error", "Please select a student.", parent=form)
                return

            # Resolve subject selection: expect subject name from combo or typed
            subject_key = (subject_var.get() or subject_combo.get() or "").strip()
            if not subject_key and getattr(subject_combo, '__getitem__', None):
                try:
                    vals = subject_combo['values']
                    if vals:
                        subject_key = vals[0]
                except Exception:
                    pass
            # Map typed/selected subject name to id
            subject_id = subject_map.get(subject_key)
            if not subject_id:
                messagebox.showerror("Error", "Please select a subject.", parent=form)
                return
            # Parse numeric inputs (support comma decimals) and read directly from Entry if needed
            raw_marks = (marks_var.get() or marks_entry.get() or "").strip()
            raw_total = (total_var.get() or total_entry.get() or "").strip()
            try:
                marks = float(raw_marks.replace(',', '.'))
                total = float(raw_total.replace(',', '.'))
            except Exception:
                messagebox.showerror("Error", "Marks and Total must be numeric.", parent=form)
                return
            if total <= 0:
                messagebox.showerror("Error", "Total must be greater than 0.", parent=form)
                return
            if marks < 0 or marks > total:
                messagebox.showerror("Error", "Marks must be between 0 and Total.", parent=form)
                return
            # Normalize date: support YYYY-MM-DD, DD-MM-YYYY, DD/MM/YYYY, YYYY/MM/DD
            exam_date = (date_var.get() or date_entry.get() or today_str).strip()
            if not exam_date:
                exam_date = today_str
            if len(exam_date) == 10:
                if exam_date[2] in ('-', '/') and exam_date[5] in ('-', '/'):
                    # Possibly DD-MM-YYYY or DD/MM/YYYY
                    try:
                        d, m, y = exam_date.replace('/', '-').split('-')
                        int(d); int(m); int(y)
                        exam_date = f"{y}-{m.zfill(2)}-{d.zfill(2)}"
                    except Exception:
                        pass
                elif exam_date[4] in ('-', '/') and exam_date[7] in ('-', '/'):
                    # YYYY-MM-DD or YYYY/MM/DD -> normalize hyphens
                    exam_date = exam_date.replace('/', '-')
            ok = db.add_mark(
                student_id,
                subject_id,
                self.teacher_profile['teacher_id'],
                marks,
                total,
                exam_date,
            )
            if ok:
                messagebox.showinfo("Success", "Mark added successfully.", parent=form)
                self.load_marks()
                self.load_dashboard_data()
                form.destroy()
            else:
                messagebox.showerror("Error", "Failed to add mark.", parent=form)

        ttk.Button(form, text="Add", command=submit).grid(row=row, column=0, columnspan=2, pady=16)
        form.transient(self.root)
        form.grab_set()
        self.root.wait_window(form)

    def edit_mark(self):
        """Edit selected mark"""
        selection = self.marks_tree.selection()
        if not selection:
            messagebox.showwarning("Warning", "Please select a mark to edit")
            return
```

```
iid = selection[0]
values = self.marks_tree.item(iid)['values']
# We need mark_id; store it in iid when loading. If iid is not numeric, editing can't proceed.
try:
    mark_id = int(iid)
except Exception:
    messagebox.showerror("Error", "Cannot edit this mark (missing identifier).")
    return

form = tk.Toplevel(self.root)
form.title("Edit Mark")
form.geometry("420x300")
form.resizable(False, False)

marks_var = tk.StringVar(value=str(values[3]))
total_var = tk.StringVar(value=str(values[4]))
date_var = tk.StringVar(value=str(values[6]))

row = 0
def add_row(label_text, widget):
    ttk.Label(form, text=label_text).grid(row=row, column=0, padx=16, pady=8, sticky="w")
    widget.grid(row=row, column=1, padx=16, pady=8, sticky="w")
    return 1

row += add_row("Marks Obtained", ttk.Entry(form, textvariable=marks_var, width=28))
row += add_row("Total Marks", ttk.Entry(form, textvariable=total_var, width=28))
# Date row with calendar icon button in edit dialog
date_entry = ttk.Entry(form, textvariable=date_var, width=28)
# Ensure default today if empty (var) and entry visibly shows a value
if not (date_var.get() or "").strip():
    date_var.set(datetime.date.today().strftime('%Y-%m-%d'))
if not (date_entry.get() or "").strip():
    date_entry.insert(0, date_var.get())
# Place label, entry, and icon
ttk.Label(form, text="Exam Date (YYYY-MM-DD)").grid(row=row, column=0, padx=16, pady=8, sticky="w")
date_entry.grid(row=row, column=1, padx=16, pady=8, sticky="w")
def set_today_edit():
    date_var.set(datetime.date.today().strftime('%Y-%m-%d'))
    try:
        date_entry.icursor('end')
    except Exception:
        pass
calendar_btn = ttk.Button(form, text="■", width=3, command=set_today_edit)
calendar_btn.grid(row=row, column=2, padx=(0, 0), pady=8, sticky="w")
row += 1

def submit():
    # Parse numeric inputs (support comma decimals)
    raw_marks = (marks_var.get() or "").strip()
    raw_total = (total_var.get() or "").strip()
    try:
        marks = float(raw_marks.replace(',', '.'))
        total = float(raw_total.replace(',', '.'))
    except Exception:
        messagebox.showerror("Error", "Marks and Total must be numbers.", parent=form)
        return
    # Normalize date similar to add flow
    exam_date = (date_var.get() or "").strip()
    if not exam_date:
        exam_date = datetime.date.today().strftime('%Y-%m-%d')
    if len(exam_date) == 10:
        if exam_date[2] in ('-', '/') and exam_date[5] in ('-', '/'):
            # DD-MM-YYYY or DD/MM/YYYY -> YYYY-MM-DD
            try:
                d, m, y = exam_date.replace('/', '-').split('-')
                int(d); int(m); int(y)
                exam_date = f"{y}-{m.zfill(2)}-{d.zfill(2)}"
            except Exception:
                pass
        elif exam_date[4] in ('-', '/') and exam_date[7] in ('-', '/'):
            # YYYY-MM-DD or YYYY/MM/DD -> normalize separators
            exam_date = exam_date.replace('/', '-')
    if db.update_mark(mark_id, marks, total, exam_date):
        messagebox.showinfo("Success", "Mark updated successfully.", parent=form)
        self.load_marks()
        # Reselect updated row for user feedback
        try:
            self.marks_tree.selection_set(str(mark_id))
            self.marks_tree.see(str(mark_id))
        except Exception:
            pass
        self.load_dashboard_data()
        form.destroy()
    else:
        messagebox.showerror("Error", "Failed to update mark.", parent=form)
ttk.Button(form, text="Update", command=submit).grid(row=row, column=0, columnspan=2, pady=16)
```

```
    form.transient(self.root)
    form.grab_set()
    self.root.wait_window(form)

def delete_mark(self):
    """Delete selected mark"""
    selection = self.marks_tree.selection()
    if not selection:
        messagebox.showwarning("Warning", "Please select a mark to delete")
        return
    iid = selection[0]
    try:
        mark_id = int(iid)
    except Exception:
        messagebox.showerror("Error", "Cannot delete this mark (missing identifier).")
        return
    if messagebox.askyesno("Confirm", "Are you sure you want to delete this mark?"):
        if db.delete_mark(mark_id):
            messagebox.showinfo("Success", "Mark deleted successfully")
            self.load_marks()
            self.load_dashboard_data()
        else:
            messagebox.showerror("Error", "Failed to delete mark")

def logout(self):
    """Logout and return to login"""
    if messagebox.askyesno("Logout", "Are you sure you want to logout?"):
        self.root.destroy()
        import login
        login.LoginWindow().run()
```

## Setup Script (setup.py)

```
#!/usr/bin/env python3
"""
Setup script for Student Performance Monitoring System
"""

import subprocess
import sys
import os
import mysql.connector
from mysql.connector import Error

def check_python_version():
    """Check if Python version is compatible"""
    if sys.version_info < (3, 7):
        print("■ Python 3.7 or higher is required")
        return False
    print(f"■ Python {sys.version_info.major}.{sys.version_info.minor} detected")
    return True

def install_dependencies():
    """Install required Python packages"""
    print("■ Installing Python dependencies...")

    try:
        subprocess.check_call([sys.executable, "-m", "pip", "install", "-r", "requirements.txt"])
        print("■ Dependencies installed successfully")
        return True
    except subprocess.CalledProcessError as e:
        print(f"■ Failed to install dependencies: {e}")
        return False

def check_mysql_connection():
    """Check MySQL connection"""
    print("■ Checking MySQL connection...")

    try:
        connection = mysql.connector.connect(
            host='localhost',
            user='root',
            password='',
            charset='utf8mb4',
            collation='utf8mb4_unicode_ci'
        )

        if connection.is_connected():
            print("■ MySQL connection successful")
            connection.close()
            return True
    except Error as e:
        print(f"■ MySQL connection failed: {e}")
        print("   Please ensure XAMPP is running and MySQL service is started")
        return False

def create_database():
    """Create database and tables"""
    print("■■ Setting up database...")

    try:
        # Connect to MySQL
        connection = mysql.connector.connect(
            host='localhost',
            user='root',
            password='',
            charset='utf8mb4',
            collation='utf8mb4_unicode_ci'
        )

        cursor = connection.cursor()

        # Read and execute database setup script
        with open('database_setup.sql', 'r', encoding='utf-8') as file:
            sql_commands = file.read()

        # Split commands and execute
        commands = sql_commands.split(';')
        for command in commands:
            command = command.strip()
            if command and not command.startswith('--'):
                try:
                    cursor.execute(command)
                    connection.commit()
                except Error as e:
                    if "already exists" not in str(e).lower():
```

```
                        print(f"■■ Warning: {e}")

            cursor.close()
            connection.close()
            print("■ Database setup completed")
            return True

        except Error as e:
            print(f"■ Database setup failed: {e}")
            return False
        except FileNotFoundError:
            print("■ database_setup.sql file not found")
            return False

    def main():
        """Main setup function"""
        print("■ Student Performance Monitoring System Setup")
        print("=" * 50)

        # Check Python version
        if not check_python_version():
            return

        # Install dependencies
        if not install_dependencies():
            return

        # Check MySQL connection
        if not check_mysql_connection():
            return

        # Create database
        if not create_database():
            return

        print("\n■ Setup completed successfully!")
        print("\n■ Next steps:")
        print("   1. Run: python main.py")
        print("   2. Use demo credentials:")
        print("      - Admin: admin / admin123")
        print("      - Teacher: teacher1 / teacher123")
        print("      - Student: student1 / student123")

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