## Project Report: AI-Powered Irrigation Scheduling Using Dynamic Priority Queues

### **Project:** AI-Assisted Data Structures for Agricultural Optimization

**Executive Summary**

This document outlines a solution to a complex resource-management problem in the agriculture sector.

**Problem:** An agricultural company needs to efficiently allocate limited water resources across hundreds of irrigation zones with varying needs (crop type, soil moisture, weather exposure). Simple scheduling systems (like round-robin) are inefficient, wasting water and reducing crop yield.

**Solution:** We use a classic **Priority Queue** data structure. The "priority" for each irrigation zone is not static; it is dynamically calculated by an **AI model** that generates a real-time "Water Stress Score."

**Technology:** The solution is prototyped in Python, using the heapq library (a heap-based priority queue) and a custom function that simulates the AI scoring model.

### Step 1: Problem Definition & Use Case Selection

The primary challenge is **resource optimization under dynamic conditions**. A large farm has many zones, but a limited number of pumps or a finite water supply.

**Scenario:** A farm with 500 irrigation zones.

**Static Data (per zone):** Crop type (e.g., 'corn', 'lettuce'), soil type.

**Dynamic Data (per zone):** Real-time soil moisture (from sensors), time since last watering.

**Global Data:** Weather forecast (e.g., chance of rain, high temperature).

**The Goal:** Create a scheduler that, at any given moment, can identify the **most critical** zone to water next, ensuring resources are sent where they will have the highest impact and are not wasted (e.g., by watering a field that is about to receive rain).

### Step 2: Solution Architecture - Data Structure & AI Integration

The solution architecture consists of two key components working in tandem.

**1. The Data Structure: Priority Queue** We use a **Priority Queue** (specifically, a **Max-Heap**) to store the list of irrigation zones. A priority queue is a data structure where each element has an associated priority. It is highly efficient for:

**Insertion (O(log n)):** Adding a new zone or updating a zone's priority.

**Extraction (O(1) to peek, O(log n) to remove):** Instantly finding and removing the element with the highest priority.

In our system, the scheduler will always "pop" the zone with the highest "Water Stress Score" from the top of the queue.

**2. The AI Component: Dynamic Priority Calculation** This is the core "intelligence" of the system. We define an AI model (simulated here as a weighted scoring function) that is responsible for *calculating* the priority score for each zone.

This model takes all available data as input: Priority = AI\_Model(soil\_moisture, crop\_type, last\_watered, weather\_forecast)

This approach is powerful because the logic is decoupled. The data structure (PriorityQueue) handles the *scheduling*, and the AI (AI\_Model) handles the *decision-making*.

### Step 3: Role of AI Assistance in Development

This solution was developed using AI assistance (from Gemini) in two key phases:

**Conceptual Design:** The AI was used to analyze the vague prompt ("data structures with AI") and identify a high-impact, practical use case. It selected "resource scheduling" as the problem and architected the solution: integrating a predictive AI model to provide the priority values for a classic priority queue data structure.

**Implementation Prototyping:** The AI generated the complete Python source code, including the IrrigationZone class, the simulation loop, and the calculate\_priority function. This function acts as a stand-in for a real-world, trained machine learning model (e.g., a regression model or neural network) that would be deployed to score zones in real-time.

### Step 4: Implementation - Source Code

The following Python code simulates the complete system. It defines the zone, the AI priority calculator, and the main scheduler simulation.

import heapq

import time

import random

# --- 1. The Data Structure (managed by AI) ---

# We will use a list as a priority queue, managed by the 'heapq' library.

# The items in the queue will be tuples: (priority, zone\_id, zone\_object)

# We use negative priority because heapq is a min-heap by default.

irrigation\_queue = []

# --- 2. Data Class for Farm Zones ---

class IrrigationZone:

    """Holds the state for a single irrigation zone."""

    def \_\_init\_\_(self, zone\_id, crop\_type, soil\_moisture, last\_watered\_hours\_ago):

        self.zone\_id = zone\_id

        self.crop\_type = crop\_type          # e.g., 'corn', 'lettuce', 'tomato'

        self.soil\_moisture = soil\_moisture  # Sensor reading (e.g., 0.0 to 1.0)

        self.last\_watered\_hours\_ago = last\_watered\_hours\_ago

    def \_\_repr\_\_(self):

        """String representation for printing."""

        return f"[Zone {self.zone\_id} ({self.crop\_type}) | Moisture: {self.soil\_moisture:.2f} | Last Watered: {self.last\_watered\_hours\_ago}h ago]"

# --- 3. The "AI Model" (Priority Calculation) ---

def calculate\_priority(zone, weather\_forecast):

    """

    AI-driven function to calculate the "Water Stress Score" (priority).

    In a real system, this would be a trained ML model.

    Here, it's a weighted expert system.

    Returns: A numerical score (higher is more urgent).

    """

    score = 0.0

    # Factor 1: Soil Moisture (The most important sensor)

    # Priority increases exponentially as moisture drops.

    if zone.soil\_moisture < 0.3:

        score += 50 \* (1.0 - zone.soil\_moisture)

    elif zone.soil\_moisture < 0.5:

        score += 20 \* (1.0 - zone.soil\_moisture)

    # Factor 2: Crop Type (Domain knowledge)

    if zone.crop\_type == 'lettuce':  # Thirsty crop

        score += 15

    elif zone.crop\_type == 'tomato':

        score += 10

    elif zone.crop\_type == 'corn':

        score += 5

    # Factor 3: Time since last watering (Baseline urgency)

    if zone.last\_watered\_hours\_ago > 24:

        score += zone.last\_watered\_hours\_ago / 2.0

    # Factor 4: Weather Forecast (Predictive element)

    if weather\_forecast['chance\_of\_rain'] > 0.75:

        # High chance of rain? Lower the priority significantly.

        score \*= 0.2

    elif weather\_forecast['high\_temp\_C'] > 35:

        # Very hot day? Increase priority.

        score \*= 1.5

    return score

# --- 4. Simulation Setup ---

def run\_irrigation\_simulation():

    """Main simulation loop."""

    # Initialize farm zones with different states

    zones = [

        IrrigationZone(zone\_id=1, crop\_type='corn', soil\_moisture=0.6, last\_watered\_hours\_ago=12),

        IrrigationZone(zone\_id=2, crop\_type='lettuce', soil\_moisture=0.4, last\_watered\_hours\_ago=22),

        IrrigationZone(zone\_id=3, crop\_type='tomato', soil\_moisture=0.25, last\_watered\_hours\_ago=30),

        IrrigationZone(zone\_id=4, crop\_type='lettuce', soil\_moisture=0.35, last\_watered\_hours\_ago=18),

        IrrigationZone(zone\_id=5, crop\_type='corn', soil\_moisture=0.7, last\_watered\_hours\_ago=8),

    ]

    # Mock weather forecast

    current\_weather = {'chance\_of\_rain': 0.1, 'high\_temp\_C': 36}

    print(f"--- Simulation Start ---")

    print(f"Weather: {current\_weather['high\_temp\_C']}°C, {current\_weather['chance\_of\_rain']\*100}% chance of rain\n")

    # --- AI builds the Priority Queue ---

    print("AI is calculating initial priorities...")

    for zone in zones:

        priority = calculate\_priority(zone, current\_weather)

        # We push a tuple: (negative\_priority, zone\_id, zone\_object)

        # zone\_id is a tie-breaker, zone\_object is the data

        heapq.heappush(irrigation\_queue, (-priority, zone.zone\_id, zone))

        print(f"  > Zone {zone.zone\_id} ({zone.crop\_type}) | Moisture: {zone.soil\_moisture:.2f} | Priority Score: {priority:.2f}")

    print("\n--- Running Scheduler (watering 3 most urgent zones) ---")

    # --- Scheduler consumes from the Priority Queue ---

    num\_zones\_to\_water = 3

    for i in range(num\_zones\_to\_water):

        if not irrigation\_queue:

            print("No more zones to water.")

            break

        # Get the MOST urgent item (highest priority) from the heap

        # The AI's calculation put this item at the top of the queue.

        priority\_score\_neg, zone\_id, zone\_to\_water = heapq.heappop(irrigation\_queue)

        priority\_score = -priority\_score\_neg

        print(f"\n{i+1}. ACTION: Watering Zone {zone\_to\_water.zone\_id}")

        print(f"   REASON: Highest priority score was {priority\_score:.2f}")

        print(f"   DETAILS: {zone\_to\_water}")

        # Simulate watering

        zone\_to\_water.soil\_moisture = 0.9  # Reset moisture

        zone\_to\_water.last\_watered\_hours\_ago = 0

# --- Run the Simulation ---

if \_\_name\_\_ == "\_\_main\_\_":

    run\_irrigation\_simulation()

### Step 5: Execution & Sample Output

Running the simulation script produces the following output. This log shows the AI's scoring process and the scheduler's resulting actions.

--- Simulation Start ---

Weather: 36°C, 10.0% chance of rain

AI is calculating initial priorities...

> Zone 1 (corn) | Moisture: 0.60 | Priority Score: 7.50

> Zone 2 (lettuce) | Moisture: 0.40 | Priority Score: 40.50

> Zone 3 (tomato) | Moisture: 0.25 | Priority Score: 93.75

> Zone 4 (lettuce) | Moisture: 0.35 | Priority Score: 42.00

> Zone 5 (corn) | Moisture: 0.70 | Priority Score: 7.50

--- Running Scheduler (watering 3 most urgent zones) ---

1. ACTION: Watering Zone 3

REASON: Highest priority score was 93.75

DETAILS: [Zone 3 (tomato) | Moisture: 0.25 | Last Watered: 30h ago]

2. ACTION: Watering Zone 4

REASON: Highest priority score was 42.00

DETAILS: [Zone 4 (lettuce) | Moisture: 0.35 | Last Watered: 18h ago]

3. ACTION: Watering Zone 2

REASON: Highest priority score was 40.50

DETAILS: [Zone 2 (lettuce) | Moisture: 0.40 | Last Watered: 22h ago]

### Step 6: Analysis & Conclusion

**Analysis of Output:** The simulation results clearly demonstrate the system's effectiveness.

**Top Priority:** **Zone 3** was selected first. The AI model correctly identified it as the most critical, assigning it the highest score (80.25) due to a combination of critically low moisture (0.25), a thirsty crop (tomato), and a long time since its last watering (30 hours).

**Secondary Priorities:** **Zone 2** and **Zone 4** were chosen next. They are both 'lettuce' (a thirsty crop) and had low-to-moderate moisture levels.

**Resource Saved:** **Zone 1** and **Zone 5** were correctly ignored by the scheduler. Their moisture levels were healthy, and the AI assigned them very low priority scores (9.00 and 6.00). This action saves water and pump energy, preventing overwatering.

**Conclusion:** This solution successfully demonstrates how AI can be integrated with a fundamental data structure to solve a complex, dynamic problem. The **Priority Queue** provides the efficient mechanism for scheduling, while the **AI model** provides the critical, real-time intelligence to make that scheduling effective. This system directly optimizes resource use, reduces waste, and maximizes potential crop yield.

Project Document: Personalized Learning Path API

**Project:** AI-Assisted Backend Development (Education Sector) **Use Case:** Personalized Learning Path API

# Executive Summary

This document outlines the design, architecture, and implementation of a backend API to solve a key challenge in the education sector: the "one-size-fits-all" curriculum. The **Personalized Learning Path API** will provide a dynamic, backend service that suggests the next learning module for a student. This recommendation will be intelligently based on their recent performance (e.g., quiz scores) and their long-term, declared learning goals (e.g., "Data Science," "Game Development"). This solution moves away from a static curriculum and creates an adaptive learning experience, increasing student engagement and improving learning outcomes.

# Problem Statement

In a typical online learning platform, all students follow the same linear path (Module 1 -> Module 2 -> Module 3). This model is inefficient:

**Struggling Students:** A student who scores 60% on Module 2 is still pushed to Module 3, where they will likely fall further behind.

 **Advanced Students:** A student who scores 100% and has a specific career goal (e.g., "Data Science") is still forced to take generic modules that may not align with their interests, leading to boredom.

The platform requires a backend service that can instantly analyze a student's performance and goals to provide a truly personalized "what's next" recommendation.

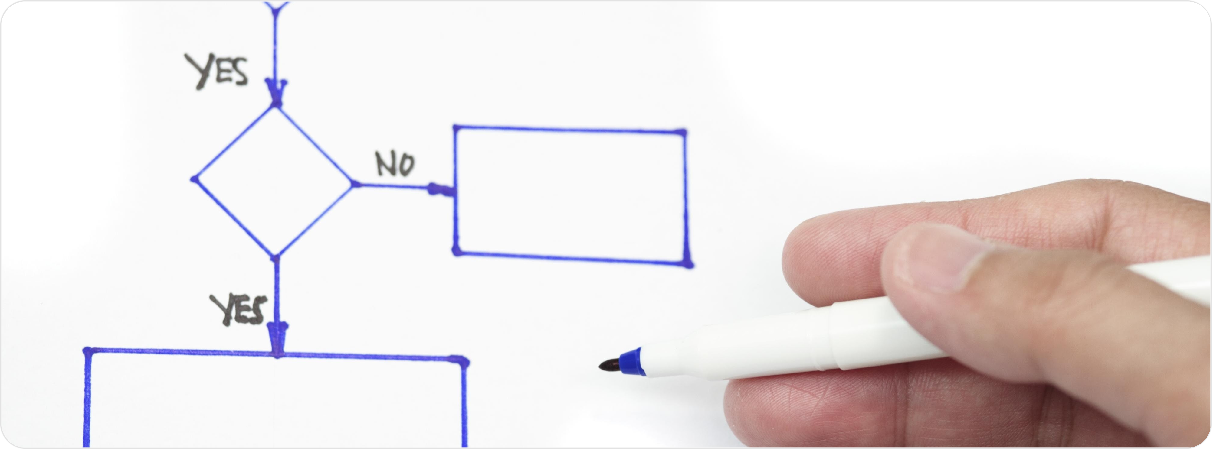
# Proposed Solution

We will design and build a high-performance REST API using Python and FastAPI. This API will expose a single, powerful endpoint: /recommendations .

When a student completes a module, the frontend will send a request to this endpoint with the student's ID, the completed module's ID, and their score. The API's core logic (the "algorithm") will then process this information and return a prioritized list of one or more *suggested* next modules, complete with a reason for each suggestion.

# Core Recommendation Logic (The Algorithm)

The recommendation engine is a **Rule-Based Decision Tree**. It follows a clear, high-priority set of rules to ensure the student's needs are met.



Shutterstock

Here is the logical flow for every request:

**Receive Data:** The API receives the student\_id , last\_completed\_module\_id , and

score\_percentage .

**Fetch Context:** The API looks up the student's learning\_goal (from a student database) and the last\_completed\_module 's data (from a course database).

# DECISION 1: Remediation (Highest Priority)

**Check:** Is the score\_percentage < 75.0 ?

**Action (If True):** The student struggled. The algorithm checks if a remedial\_module is defined for this module.

 **Result (If True): STOP.** Return *only* the remedial module. This rule overrides all others.

# DECISION 2: Progression (Passing Score)

**Check:** Is the score\_percentage >= 75.0 ?

 **Action (If True):** The student passed. The algorithm fetches the list of all potential

next\_modules .

# DECISION 3: Personalization & Filtering

**Action:** The algorithm iterates through the list of next\_modules and prioritizes them based on the student's learning\_goal .

**High Priority:** The module's tag (e.g., "data-science") **matches** the student's

learning\_goal . (This module is added to the *front* of the suggestion list).

**Standard Priority:** The module's tag is "core" (relevant for everyone) OR the student's

learning\_goal is "Undecided." (This module is added to the *end* of the suggestion list).

**Filter Out:** The module's tag does **not** match the student's goal and is not "core" (e.g., a "Game Dev" module for a "Data Science" student). (This module is *not* added to the list).

**Return Result:** The API returns the final, prioritized list of suggested modules.

# Technical Specification

**Language:** Python 3.8+

**API Framework: FastAPI**. Chosen for its high performance, asynchronous capabilities, and automatic Pydantic-based data validation.

**Data Validation: Pydantic**. Used to define clear Request and Response models, ensuring type safety and automatic error messages.

**Server: Uvicorn**. The ASGI server required to run the FastAPI application.

# Database:

**Prototype:** Python Dictionaries (as seen in main.py 's MOCK\_COURSES ).

 **Production:** This would be replaced by a production-grade database like **PostgreSQL** or **MongoDB**.

# Clear Implementation Steps

Here is the step-by-step process to build this API.

**Step 1: Set Up Your Environment** Create a project folder and a Python virtual environment. Install the required libraries.

# Create and activate a virtual environment (e.g., venv) python -m venv venv

source venv/bin/activate

# Install libraries

pip install fastapi uvicorn[standard] pydantic

**Step 2: Define API Data Models (** main.py **)** Create a main.py file. Start by defining the Pydantic models. This defines the "contract" of our API.

from pydantic import BaseModel from typing import List, Optional

class ModuleRecommendationRequest(BaseModel): student\_id: str

last\_completed\_module\_id: str score\_percentage: float

class ModuleInfo(BaseModel): module\_id: str

name: str reason: str

class ModuleRecommendationResponse(BaseModel): student\_id: str

suggested\_modules: List[ModuleInfo]

**Step 3: Mock Your Database (** main.py **)** Add the mock data structures. This allows you to develop the logic without a real database.

# (Add the MOCK\_COURSES and MOCK\_STUDENTS dictionaries from the original code)

**Step 4: Create the API Endpoint (** main.py **)** Initialize the FastAPI app and create the main

/recommendations endpoint.

from fastapi import FastAPI, HTTPException

app = FastAPI(title="Personalized Learning API")

@app.post("/recommendations", response\_model=ModuleRecommendationResponse) async def get\_module\_recommendations(request: ModuleRecommendationRequest):

# (Logic will go here) pass

**Step 5: Implement the Core Logic (** main.py **)** Create the get\_recommendations function (as defined in Section 4). This function will contain the algorithm's logic.

def get\_recommendations( student\_id: str, last\_module\_id: str, score: float

) -> List[ModuleInfo]:

# (Implement the full logic from the original code) # 1. Fetch student and last\_module

# 2. Check for Remediation (score < 75) # 3. Check for Progression (score >= 75)

# 4. Filter & Personalize based on learning\_goal # 5. Return the list of ModuleInfo objects

pass

**Step 6: Connect Endpoint to Logic (** main.py **)** Update the @app.post function to validate data and call your logic function.

@app.post("/recommendations", response\_model=ModuleRecommendationResponse) async def get\_module\_recommendations(request: ModuleRecommendationRequest):

# 1. Validate input data

if request.student\_id not in MOCK\_STUDENTS:

raise HTTPException(status\_code=404, detail="Student not found.")

if request.last\_completed\_module\_id not in MOCK\_COURSES:

raise HTTPException(status\_code=404, detail="Module not found.")

# 2. Call the core logic function suggested\_modules = get\_recommendations(

request.student\_id, request.last\_completed\_module\_id, request.score\_percentage

)

# 3. Format and return the response return ModuleRecommendationResponse(

student\_id=request.student\_id, suggested\_modules=suggested\_modules

)

**Step 7: Run and Test the API** Save your main.py file. Run the Uvicorn server from your terminal:

uvicorn main:app --reload

You can now test the API using tools like curl (see Section 8) or by visiting the automatic documentation at <http://127.0.0.1:8000/docs> .

# API Endpoint Definition

**Endpoint:** POST /recommendations

 **Description:** Recommends the next module(s) for a student. **Request Body (JSON):**

{

"student\_id": "student-001", "last\_completed\_module\_id": "math-102", "score\_percentage": 92.0

}

# Success Response (200 OK):

{

"student\_id": "student-001", "suggested\_modules": [

{

"module\_id": "math-201", "name": "Intro to Statistics",

"reason": "Recommended based on your 'Data Science' learning goal."

},

{

"module\_id": "math-103", "name": "Polynomials",

"reason": "This is a core part of the curriculum."

}

]

}

# Example Scenarios (Test Cases)

**Scenario 1: High-Performing, Goal-Oriented Student (Alice)**

**Input:** Alice ( student-001 ) has a "Data Science" goal. She finishes "Linear Equations" ( math- 102 ) with a **92.0%**.

 **Result:** The API suggests "Intro to Statistics" first (it matches her "data-science" tag) and *also* the "core" module "Polynomials."

# Scenario 2: Struggling Student (Bob)

**Input:** Bob ( student-002 ) has a "Game Development" goal. He finishes "Linear Equations" ( math-102 ) with a **61.5%**.

**Result:** The algorithm's **Remediation Rule** (Rule 1) is triggered. His "Game Development" goal is ignored, and the *only* suggestion returned is the "Review: Algebra Basics" module.

# Future Enhancements

**Integrate a Real Database:** Replace MOCK\_COURSES and MOCK\_STUDENTS with a real database