太好了！基于你给出的算法 outline（我已阅读）和你们的整体 Hackathon 计划，我整理了一份\*\*“算法负责人全流程作战大纲 + Checklist”\*\*。它兼容你现有的数据结构与函数切分，并在关键节点给出可执行/可验收的标准与降级方案。文中引用你文件里的关键片段做对齐。

# **目标与边界**

* **MVP 能力**：  
  1. 依据课表 + 当前时间计算用户状态（in\_class / free / 手动状态覆盖）；
  2. 学习伙伴推荐（同课 + 空闲 + 简单偏好兼容）；
  3. 可选：社交匹配与「同课连接/下一节同课」增强项。
* **不做**：训练真实模型、复杂个性化特征学习（保留文案与接口即可，演示时作为展望）。
* **与前后端契约**：提供 /recommendations 之类接口输出「候选 + 可解释 reason」；前端每 30 秒轮询（由前端负责）【“Auto-refresh every 30 seconds” 与“find study partners 接口”等在大纲中已给出】。

# **一、数据与基础设施（T+0～2h）**

## **1) 数据文件与装载**

* 路径常量：DATA\_DIR / users.json / schedules.json / status.json（已在 outline 里定义）
* 装载方法：load\_json\_file() 带容错（文件不存在/JSON 解析失败返回空 dict）
* 引擎类：RecommendationEngine 含 refresh\_data()，在每次推荐/查询前刷新一次，保证演示期间热更新

**Checklist**

* engine = RecommendationEngine() 全局单例初始化
* 所有外部函数第一行 engine.refresh\_data()
* 单元测试：空文件、格式错误文件时不崩（返回空结果）

## **2) 基础查询工具**

* get\_user\_classes(user\_id)：返回课程 code 列表
* get\_shared\_classes(u1, u2)：集合交集返回列表
* get\_user\_availability(user\_id)：从 status.json 推断“当前可用性 + 偏好标签”（studying/help → study\_high；tired/busy → social\_low；social → social\_high）

**Checklist**

* 未设置状态时返回 {status: 'unknown', social\_preference:'neutral', ...}（保证鲁棒）
* 针对 manualStatus 的枚举映射有单元测试（包括非法值）

# **二、时间与状态判定（T+2～4h）**

## **1) 时间工具**

* time\_to\_minutes("HH:MM") → int；出错返回 0（防御式）
* is\_time\_between(current, start, end)：闭区间判断（演示对齐）

## **2) 用户当前状态**

* get\_current\_class\_for\_user(user\_id)：按**今天**星期几 + 时段判断是否 “In {course} / Free”
* get\_next\_class\_for\_user(user\_id)：今天下一门课（最早 startTime > now）

**Checklist**

* 边界时刻（start==now / end==now）覆盖；
* 今天无课/已过最后一节 → 返回 Free / None；
* 演示钩子：允许 now 通过后端查询参数注入（由后端支持），算法函数也接受可选 now 便于回放演示。

# **三、学习伙伴推荐（MVP 核心，T+4～7h）**

## **1) 主函数**

* get\_study\_partners(user\_id)：你已写的主体思路正确，建议固定以下**打分项**（总分 0~∞，排序取 Top-N）：  
  + 共享课程：+10 \* len(shared\_classes)（最高权重）
  + 同专业：+5（弱关联）
  + 双方学习偏好高：+8（studying/help → study\_high）
  + 对方能帮忙（other.status == 'help'）：+6
  + 对方疲惫/忙碌：-5（过滤或惩罚其一即可）
  + 仅当 score>0 才入候选，返回 username/major/shared\_classes/score/reason(前两条)

**Checklist**

* engine.refresh\_data()；user\_id 不存在→空列表
* reasons 至少包含一条可读句子（如 “Shares 2 classes with you; Both interested in studying”）
* Top-N（默认 5）且稳定排序（分数相同按 username 次序）

# **四、社交匹配（可选增强，T+7～9h）**

* get\_social\_matches(user\_id)：仅当用户 social\_preference != 'low' 时执行；规则与你 outline 基本一致（双方都想社交 + 当前可用 + 同专业/同课加分，疲惫/忙碌/学习中强惩罚）
* 返回 Top-5，附两条 reason 文案

**Checklist**

* 学习与社交结果结构一致，便于前端统一渲染
* 当用户表达“低社交意愿”直接返回空列表（对齐你代码）

# **五、同课连接 / 下一节同课（可选增强，T+9～12h）**

* get\_class\_connections(user\_id)：  
  + 若当前就在同一门课：**+15**（最高优先）并给出 “Currently in {class}” 的理由；
  + 刚上完同一门课（近 1 小时内，可简化为“今天在课表里且早于当前时刻”）：**+8**；
  + 下一节将上同一门课：**+10**，理由 “Next class: {code}”。  
     以上逻辑与你 outline 保持一致（得分/理由模版）

**Checklist**

* 优先保证“当前同课”排最前（权重足够大）
* 无候选时返回空数组而非 None
* 理由字段包含 current\_class/next\_class 的简短说明（见 outline 返回结构）

# **六、洞察（锦上添花，T+12～15h）**

* get\_user\_insights(user\_id) 汇总学习/社交/课表效率等 3 类洞察，截前 5 条，内部可先做**规则化**：  
  + 学习负载：课程数量 ≥3 给出组队建议（示例已在 outline）
  + 常见时段：根据最近 N 次手动状态 studying 的时间段，产出“你在 Tue 14–16 学习效率高”；
  + 关系线索：与某人频繁同课/同空档 → 提示“你与 Alice 在周二常同时空闲”；
* 注意：评委更看重可解释性而非复杂度（大纲也强调“Smart recommendations + Insights”的演示节奏）

**Checklist**

* 至少 2 条稳定可复现的洞察（不依赖长历史）
* 文案不含空字段；无数据时返回空数组

# **七、接口与数据契约（与后端/前端对齐，T+15～16h）**

* 输出结构与你们前端/后端约定一致：候选 id/username/major/score/reason，可选 shared\_classes/current\_class/next\_class 字段（你在 Outline 里已提供示例返回）
* 支持三类查询路由（后端负责路由到这些函数）：  
  + /recommendations?user\_id=...（学习伙伴）【大纲中已定义】
  + /social?user\_id=...（社交匹配，可选）
  + /class\_connections?user\_id=...（同课连接，可选）

**Checklist**

* 任何输入缺失/用户不存在 → 200 + 空数组（避免 5xx）
* 统一字段命名与大小写（与现有 users/schedules/status 对齐）

# **八、种子数据与可复现实验（T+16～18h）**

* seed.py 生成 10–15 人假数据（你的 Outline 已包含范例结构：三位用户的课程表、若干 manualStatus）
* 保证至少：  
  + 有两门高重叠课程（便于“同课 + 空闲”命中）；
  + 多样的 manualStatus：studying/free/social/tired/help（触发不同规则）

**Checklist**

* 一键写入 users/schedules/status 三个 JSON（已在你 outline 的保存函数中示例化）
* 运行一次推荐函数打印 Top-3（或 Top-5）到控制台

# **九、质量保障（T+18～20h）**

## **1) 快速单元测试（脚本级）**

* **时间函数**：多个边界样例（00:00、23:59、同一时刻）
* **状态函数**：今天无课 / 正在上课 / 刚下课 / 即将上课
* **推荐函数**：  
  + 有同课但对方忙碌 → 被惩罚
  + 无同课但 help → 仍可进入候选（低分）
  + 没有任何人符合 → 返回 []

## **2) 可解释性检查**

* 每个候选 **必须**带 reason（前 1–2 条），对应规则映射（已在代码片段中体现）

# **十、演示优化与降级策略（T+20～22h）**

* **演示时钟**：优先走“下一节同课”或“当前同课”剧本（权重已高）
* **失败保护**：  
  + 推荐为空 → “暂无同课可用同学，稍后再试/切换到社交匹配”
  + 时间注入失效 → 手动将两人 manualStatus 设为 help/free 以触发学习匹配
* **日志**：在服务端打印每个候选的打分细目（便于排查）

# **十一、最终自检（T+22～24h）— 超短清单**

* get\_study\_partners、get\_social\_matches、get\_class\_connections 均可在命令行直接跑出结果（不依赖后端路由）
* Top-N 排序稳定；reason 可读；空结果不报错
* 与前端字段完全对齐；演示数据能稳定产出 ≥2 个候选
* 预留“未来展望”台词：ML 加权/个性化、跨校园扩展（契合你们的文档叙事）

## **你当前 Outline 里可直接沿用/巩固的模块**

* 数据引擎/装载与刷新（已具备）
* 时间与状态工具（已具备）
* 学习/社交/同课/洞察四大函数的雏形（已具备并可微调权重与文案）

Algorithm/ML Developer

Hours 1-2: Data Structures & Basic Logic

# algorithm.py

import json

from datetime import datetime, timedelta

import os

# Import data loading functions from backend

def load\_json\_file(filename):

"""Load JSON file or return empty dict if file doesn't exist"""

try:

with open(filename, 'r') as f:

return json.load(f)

except FileNotFoundError:

return {}

except json.JSONDecodeError:

return {}

# Data file paths

DATA\_DIR = 'data'

USERS\_FILE = os.path.join(DATA\_DIR, 'users.json')

SCHEDULES\_FILE = os.path.join(DATA\_DIR, 'schedules.json')

STATUS\_FILE = os.path.join(DATA\_DIR, 'status.json')

class RecommendationEngine:

def \_\_init\_\_(self):

self.users = load\_json\_file(USERS\_FILE)

self.schedules = load\_json\_file(SCHEDULES\_FILE)

self.statuses = load\_json\_file(STATUS\_FILE)

def refresh\_data(self):

"""Refresh data from files"""

self.users = load\_json\_file(USERS\_FILE)

self.schedules = load\_json\_file(SCHEDULES\_FILE)

self.statuses = load\_json\_file(STATUS\_FILE)

def get\_user\_classes(self, user\_id):

"""Get list of classes for a user"""

if user\_id not in self.schedules:

return []

classes = []

for class\_info in self.schedules[user\_id]['schedule']:

classes.append(class\_info['courseCode'])

return classes

def get\_shared\_classes(self, user1\_id, user2\_id):

"""Get classes shared between two users"""

user1\_classes = set(self.get\_user\_classes(user1\_id))

user2\_classes = set(self.get\_user\_classes(user2\_id))

return list(user1\_classes.intersection(user2\_classes))

def get\_user\_availability(self, user\_id):

"""Determine user's current availability"""

if user\_id not in self.statuses:

return {'status': 'unknown', 'social\_preference': 'neutral'}

manual\_status = self.statuses[user\_id].get('manualStatus', 'free')

availability = {

'status': manual\_status,

'social\_preference': 'neutral',

'study\_preference': 'neutral'

}

# Interpret status for social and study preferences

if manual\_status in ['social', 'want to hang out']:

availability['social\_preference'] = 'high'

elif manual\_status in ['tired', 'busy']:

availability['social\_preference'] = 'low'

elif manual\_status in ['studying', 'help']:

availability['study\_preference'] = 'high'

return availability

# Initialize global engine

engine = RecommendationEngine()

Hours 3-6: Study Partner Algorithm

# Add to algorithm.py

def get\_study\_partners(user\_id):

"""Find study partners for a user based on shared classes and compatibility"""

engine.refresh\_data()

if user\_id not in engine.users:

return []

user\_classes = engine.get\_user\_classes(user\_id)

user\_availability = engine.get\_user\_availability(user\_id)

user\_major = engine.users[user\_id]['major']

candidates = []

for other\_id, other\_user in engine.users.items():

if other\_id == user\_id:

continue

# Calculate compatibility score

score = 0

reasons = []

# Shared classes (highest priority)

shared\_classes = engine.get\_shared\_classes(user\_id, other\_id)

if shared\_classes:

score += len(shared\_classes) \* 10

reasons.append(f"Shares {len(shared\_classes)} classes with you")

# Same major bonus

if other\_user['major'] == user\_major:

score += 5

reasons.append("Same major")

# Availability compatibility

other\_availability = engine.get\_user\_availability(other\_id)

# Both want to study

if (user\_availability['study\_preference'] == 'high' and

other\_availability['study\_preference'] == 'high'):

score += 8

reasons.append("Both interested in studying")

# Other person can help

if (user\_availability['study\_preference'] == 'high' and

other\_availability['status'] == 'help'):

score += 6

reasons.append("Available to help with studies")

# Avoid people who are tired or busy

if other\_availability['status'] in ['tired', 'busy']:

score -= 5

# Only recommend if there's some connection

if score > 0:

candidates.append({

'id': other\_id,

'username': other\_user['username'],

'major': other\_user['major'],

'score': score,

'shared\_classes': shared\_classes,

'reason': '; '.join(reasons[:2]) # Top 2 reasons

})

# Sort by score and return top candidates

candidates.sort(key=lambda x: x['score'], reverse=True)

return candidates[:5]

def get\_social\_matches(user\_id):

"""Find social matches based on availability and preferences"""

engine.refresh\_data()

if user\_id not in engine.users:

return []

user\_availability = engine.get\_user\_availability(user\_id)

user\_major = engine.users[user\_id]['major']

# Only recommend social matches if user is interested

if user\_availability['social\_preference'] == 'low':

return []

candidates = []

for other\_id, other\_user in engine.users.items():

if other\_id == user\_id:

continue

other\_availability = engine.get\_user\_availability(other\_id)

score = 0

reasons = []

# Both want to socialize

if (user\_availability['social\_preference'] == 'high' and

other\_availability['social\_preference'] == 'high'):

score += 10

reasons.append("Both want to hang out")

# Available and not busy

if other\_availability['status'] in ['free', 'social']:

score += 5

reasons.append("Currently available")

# Same major (might have common interests)

if other\_user['major'] == user\_major:

score += 3

reasons.append("Same major")

# Some shared classes (common ground)

shared\_classes = engine.get\_shared\_classes(user\_id, other\_id)

if shared\_classes:

score += 2

reasons.append("Has classes with you")

# Avoid tired/busy people

if other\_availability['status'] in ['tired', 'busy', 'studying']:

score -= 8

if score > 0:

candidates.append({

'id': other\_id,

'username': other\_user['username'],

'major': other\_user['major'],

'score': score,

'reason': '; '.join(reasons[:2])

})

candidates.sort(key=lambda x: x['score'], reverse=True)

return candidates[:5]

Hours 7-12: Advanced Matching & Class Connections

# Add to algorithm.py

def get\_class\_connections(user\_id):

"""Find people currently in the same class or recently finished"""

engine.refresh\_data()

if user\_id not in engine.users:

return []

# Get user's current class

current\_class = get\_current\_class\_for\_user(user\_id)

candidates = []

for other\_id, other\_user in engine.users.items():

if other\_id == user\_id:

continue

other\_current\_class = get\_current\_class\_for\_user(other\_id)

score = 0

reasons = []

# Currently in the same class

if (current\_class != "Free" and current\_class == other\_current\_class):

score += 15

reasons.append(f"Currently in {current\_class.replace('In ', '')}")

# Recently finished the same class (within last hour)

elif current\_class != "Free":

class\_code = current\_class.replace('In ', '')

if class\_code in engine.get\_user\_classes(other\_id):

# Check if they had this class recently

score += 8

reasons.append(f"Recently in {class\_code}")

# About to have the same class (next class)

next\_class\_user = get\_next\_class\_for\_user(user\_id)

next\_class\_other = get\_next\_class\_for\_user(other\_id)

if (next\_class\_user and next\_class\_other and

next\_class\_user['courseCode'] == next\_class\_other['courseCode']):

score += 10

reasons.append(f"Next class: {next\_class\_user['courseCode']}")

if score > 0:

candidates.append({

'id': other\_id,

'username': other\_user['username'],

'major': other\_user['major'],

'score': score,

'current\_class': other\_current\_class,

'reason': '; '.join(reasons)

})

candidates.sort(key=lambda x: x['score'], reverse=True)

return candidates[:3]

def get\_current\_class\_for\_user(user\_id):

"""Get user's current class (same logic as backend)"""

if user\_id not in engine.schedules:

return "Free"

current\_day = datetime.now().strftime('%A').lower()

current\_time = datetime.now().strftime('%H:%M')

user\_schedule = engine.schedules[user\_id]['schedule']

for class\_info in user\_schedule:

if class\_info['day'] == current\_day:

if is\_time\_between(current\_time, class\_info['startTime'], class\_info['endTime']):

return f"In {class\_info['courseCode']}"

return "Free"

def get\_next\_class\_for\_user(user\_id):

"""Get user's next class today"""

if user\_id not in engine.schedules:

return None

current\_day = datetime.now().strftime('%A').lower()

current\_time = datetime.now().strftime('%H:%M')

current\_minutes = time\_to\_minutes(current\_time)

user\_schedule = engine.schedules[user\_id]['schedule']

next\_classes = []

for class\_info in user\_schedule:

if class\_info['day'] == current\_day:

start\_minutes = time\_to\_minutes(class\_info['startTime'])

if start\_minutes > current\_minutes:

next\_classes.append({

'courseCode': class\_info['courseCode'],

'courseName': class\_info['courseName'],

'startTime': class\_info['startTime'],

'start\_minutes': start\_minutes

})

if next\_classes:

next\_classes.sort(key=lambda x: x['start\_minutes'])

return next\_classes[0]

return None

def time\_to\_minutes(time\_str):

"""Convert HH:MM time to minutes since midnight"""

try:

hours, minutes = map(int, time\_str.split(':'))

return hours \* 60 + minutes

except:

return 0

def is\_time\_between(current, start, end):

"""Check if current time is between start and end"""

current\_min = time\_to\_minutes(current)

start\_min = time\_to\_minutes(start)

end\_min = time\_to\_minutes(end)

return start\_min <= current\_min <= end\_min

Hours 13-18: Smart Insights & Pattern Recognition

# Add to algorithm.py

def get\_user\_insights(user\_id):

"""Generate insights about user's patterns and suggestions"""

engine.refresh\_data()

if user\_id not in engine.users:

return []

insights = []

# Analyze study patterns

study\_insights = analyze\_study\_patterns(user\_id)

insights.extend(study\_insights)

# Analyze social patterns

social\_insights = analyze\_social\_patterns(user\_id)

insights.extend(social\_insights)

# Analyze schedule efficiency

schedule\_insights = analyze\_schedule\_efficiency(user\_id)

insights.extend(schedule\_insights)

return insights[:5] # Return top 5 insights

def analyze\_study\_patterns(user\_id):

"""Analyze when user studies best"""

insights = []

# Simple pattern analysis based on status updates

if user\_id in engine.statuses:

# This would be more sophisticated with historical data

user\_classes = engine.get\_user\_classes(user\_id)

if len(user\_classes) >= 3:

insights.append({

'type': 'study\_load',

'message': f"You have {len(user\_classes)} classes - consider forming study groups for better efficiency",

'priority': 'medium'

})

# Check for STEM classes (might need more study support)

stem\_classes = [c for c in user\_classes if any(stem in c.upper() for stem in ['CS', 'MATH', 'PHYS', 'CHEM', 'ENG'])]

if stem\_classes:

insights.append({

'type': 'study\_support',

'message': f"Your STEM classes ({', '.join(stem\_classes)}) often benefit from group study",

'priority': 'high'

})

return insights

def analyze\_social\_patterns(user\_id):

"""Analyze user's social interaction patterns"""

insights = []

# Analyze major-based social opportunities

user\_major = engine.users[user\_id]['major']

same\_major\_count = sum(1 for u in engine.users.values() if u['major'] == user\_major and u['id'] != user\_id)

if same\_major\_count >= 3:

insights.append({

'type': 'social\_opportunity',

'message': f"There are {same\_major\_count} other {user\_major} students - great for networking!",

'priority': 'medium'

})

return insights

def analyze\_schedule\_efficiency(user\_id):

"""Analyze schedule gaps and suggest optimizations"""

insights = []

if user\_id not in engine.schedules:

return insights

schedule = engine.schedules[user\_id]['schedule']

# Group by day

daily\_schedules = {}

for class\_info in schedule:

day = class\_info['day']

if day not in daily\_schedules:

daily\_schedules[day] = []

daily\_schedules[day].append(class\_info)

# Analyze each day

for day, classes in daily\_schedules.items():

if len(classes) >= 2:

# Sort by start time

classes.sort(key=lambda x: time\_to\_minutes(x['startTime']))

# Check for large gaps

for i in range(len(classes) - 1):

current\_end = time\_to\_minutes(classes[i]['endTime'])

next\_start = time\_to\_minutes(classes[i+1]['startTime'])

gap\_minutes = next\_start - current\_end

if gap\_minutes >= 120: # 2+ hour gap

insights.append({

'type': 'schedule\_gap',

'message': f"You have a {gap\_minutes//60}h gap on {day.title()}s - perfect for study groups!",

'priority': 'low'

})

return insights

# Advanced matching algorithm that combines all factors

def get\_comprehensive\_recommendations(user\_id):

"""Get comprehensive recommendations combining all algorithms"""

engine.refresh\_data()

all\_recommendations = []

# Get recommendations from each algorithm

study\_partners = get\_study\_partners(user\_id)

social\_matches = get\_social\_matches(user\_id)

class\_connections = get\_class\_connections(user\_id)

# Combine and deduplicate

seen\_users = set()

# Prioritize class connections (immediate relevance)

for rec in class\_connections:

if rec['id'] not in seen\_users:

rec['algorithm'] = 'class\_connection'

all\_recommendations.append(rec)

seen\_users.add(rec['id'])

# Add study partners

for rec in study\_partners:

if rec['id'] not in seen\_users:

rec['algorithm'] = 'study\_partner'

all\_recommendations.append(rec)

seen\_users.add(rec['id'])

# Add social matches

for rec in social\_matches:

if rec['id'] not in seen\_users:

rec['algorithm'] = 'social\_match'

all\_recommendations.append(rec)

seen\_users.add(rec['id'])

return all\_recommendations[:8] # Return top 8 overall

# Create sample data for demo

def create\_sample\_data():

"""Create sample users and schedules for demo purposes"""

sample\_users = {

'user1': {

'id': 'user1',

'username': 'Alice Chen',

'major': 'CS',

'created\_at': '2025-09-12T08:00:00'

},

'user2': {

'id': 'user2',

'username': 'Bob Smith',

'major': 'CS',

'created\_at': '2025-09-12T08:30:00'

},

'user3': {

'id': 'user3',

'username': 'Carol Johnson',

'major': 'MATH',

'created\_at': '2025-09-12T09:00:00'

},

'user4': {

'id': 'user4',

'username': 'David Wilson',

'major': 'CS',

'created\_at': '2025-09-12T09:30:00'

},

'user5': {

'id': 'user5',

'username': 'Emma Davis',

'major': 'PHYS',

'created\_at': '2025-09-12T10:00:00'

}

}

sample\_schedules = {

'user1': {

'userId': 'user1',

'schedule': [

{'courseCode': 'CS 15-122', 'courseName': 'Principles of Imperative Computation', 'day': 'monday', 'startTime': '09:00', 'endTime': '10:20'},

{'courseCode': 'CS 15-122', 'courseName': 'Principles of Imperative Computation', 'day': 'wednesday', 'startTime': '09:00', 'endTime': '10:20'},

{'courseCode': 'MATH 21-127', 'courseName': 'Concepts of Mathematics', 'day': 'tuesday', 'startTime': '11:30', 'endTime': '12:50'},

{'courseCode': 'MATH 21-127', 'courseName': 'Concepts of Mathematics', 'day': 'thursday', 'startTime': '11:30', 'endTime': '12:50'}

]

},

'user2': {

'userId': 'user2',

'schedule': [

{'courseCode': 'CS 15-122', 'courseName': 'Principles of Imperative Computation', 'day': 'monday', 'startTime': '09:00', 'endTime': '10:20'},

{'courseCode': 'CS 15-122', 'courseName': 'Principles of Imperative Computation', 'day': 'wednesday', 'startTime': '09:00', 'endTime': '10:20'},

{'courseCode': 'CS 15-150', 'courseName': 'Principles of Functional Programming', 'day': 'tuesday', 'startTime': '13:30', 'endTime': '14:50'}

]

},

'user3': {

'userId': 'user3',

'schedule': [

{'courseCode': 'MATH 21-127', 'courseName': 'Concepts of Mathematics', 'day': 'tuesday', 'startTime': '11:30', 'endTime': '12:50'},

{'courseCode': 'MATH 21-127', 'courseName': 'Concepts of Mathematics', 'day': 'thursday', 'startTime': '11:30', 'endTime': '12:50'},

{'courseCode': 'MATH 21-259', 'courseName': 'Calculus in Three Dimensions', 'day': 'monday', 'startTime': '14:30', 'endTime': '15:50'}

]

}

}

sample\_statuses = {

'user1': {'manualStatus': 'studying', 'lastUpdated': '2025-09-12T14:00:00'},

'user2': {'manualStatus': 'free', 'lastUpdated': '2025-09-12T14:30:00'},

'user3': {'manualStatus': 'social', 'lastUpdated': '2025-09-12T15:00:00'},

'user4': {'manualStatus': 'tired', 'lastUpdated': '2025-09-12T15:30:00'},

'user5': {'manualStatus': 'help', 'lastUpdated': '2025-09-12T16:00:00'}

}

# Save sample data

save\_json\_file(USERS\_FILE, sample\_users)

save\_json\_file(SCHEDULES\_FILE, sample\_schedules)

save\_json\_file(STATUS\_FILE, sample\_statuses)

print("Sample data created successfully!")

def save\_json\_file(filename, data):

"""Save data to JSON file"""

os.makedirs(os.path.dirname(filename), exist\_ok=True)

with open(filename, 'w') as f:

json.dump(data, f, indent=2)

# Main functions to export for backend

def main():

"""Main function for testing"""

create\_sample\_data()

# Test recommendations

recommendations = get\_comprehensive\_recommendations('user1')

print("Recommendations for Alice:")

for rec in recommendations:

print(f"- {rec['username']}: {rec['reason']} (Score: {rec['score']})")

if \_\_name\_\_ == '\_\_main\_\_':

main()