PRECISION

| **Category** | **Technology** | **Used In** | **Purpose in Your Project** |
| --- | --- | --- | --- |
| **Programming** | **Python** | Entire project (.py files) | Core language for scripting the agent logic, monitoring, decision-making, and updates |
| **AI/ML** | **Scikit-learn / TensorFlow / PyTorch** *(Optional – advanced)* | Future improvement for decision\_engine.py | For learning from real historical data and predicting optimal update times |
| **Agentic AI** | **LangChain / Autogen** *(Optional)* | Future enhancement | For creating autonomous agents that can manage updates in complex real-time environments |
| **LLMs** | **OpenAI / LLaMA / Claude** *(Optional)* | explain\_ai.py *(next step)* | To generate human-readable explanations for each decision taken by the AI |
| **Cloud Services** | **AWS / GCP / Azure** *(Optional)* | Deployment / Logging / Model hosting | To host the dashboard, models, or deploy large-scale agent infrastructure |
| **Monitoring** | **Prometheus / Grafana** *(Optional)* | Future dashboard integration | For real-time visualization of system status and update performance over time |
| **Security Tools** | **OWASP / CVE database APIs** *(Optional)* | Risk scoring logic (future version) | To fetch known vulnerabilities and assess the urgency of security updates |

**✅ Full Prototype Development Plan**

We'll build the system in **6 major steps**, each with focused modules.

**🚧 Step 1: Project Setup**

**📁 Folder Structure**

graphql

CopyEdit

smart-update-manager/

├── main.py # Core execution

├── monitor.py # System usage simulator

├── decision\_engine.py # AI decision logic

├── updater.py # Simulate applying updates

├── logger.py # Log all actions

├── config.py # Settings (e.g., thresholds)

├── explain\_ai.py (optional) # Use LLM for decision explanation

├── data/ # Mock user/system data

└── logs/ # Logs of agent behavior

**🧠 Step 2: Simulate System Monitoring**

* Create a **monitor.py** to simulate:
  + CPU usage
  + User activity (active/inactive)
  + Battery level
  + Network status

This will generate mock system status every few seconds for testing.

**🧮 Step 3: Agentic Decision Engine**

* In **decision\_engine.py**:
  + Write rules (or ML logic) to evaluate:
    - Is the system idle?
    - Is the battery > 30%?
    - Is the update urgent?
  + Output: "Apply", "Postpone", or "Modify"

Optional: Add basic **risk scoring logic** here.

**🔧 Step 4: Update Handler**

* In **updater.py**:
  + Simulate applying the update (log output and sleep)
  + Handle:
    - Postpone logic (reschedule)
    - Partial update (for low resources)

**📄 Step 5: Logging + Explainability**

* Use **logger.py** to write all actions to file
* Optional: Use **explain\_ai.py** with OpenAI to generate explanations like:

“Update postponed due to high CPU usage and user activity.”

**🖥️ Step 6: Run Everything via main.py**

* Load the mock data
* Monitor system
* Pass data to the decision engine
* Based on output, call the updater
* Log and explain everything

**🛠️ Bonus / Advanced (If time allows)**

* 📊 **Simple CLI or Streamlit dashboard**
* 📈 Visualize logs (e.g., system state over time, decisions made)
* 🧪 Add a simple ML model to improve prediction from logs
* 🌐 Host a live version (optional) on Replit, Render, or local demo

**🧪 What You’ll Show in the Demo**

* A running AI agent making **smart update decisions**
* Real-time logs showing system status, decision taken, and actions
* Optional: LLM-generated explanations for transparency
* You can now extend with:
  + OpenAI integration (for explanations)
  + Real system monitoring via psutil
  + Simple dashboard via Streamlit

**✅ 1. Predictive Analytics**

**Goal**: Use historical logs + machine learning to anticipate optimal update timing.

**Current Status**:

* ❌ Not yet implemented.
* You are logging decisions (decisions.csv), so **you have the data foundation**.
* But you're not yet using any ML model to predict when would be the best time to update.

**To achieve**:

* Train a simple model (e.g. decision tree or logistic regression) using past logs.
* Input features: battery, CPU, user activity, network, risk.
* Output: best time to apply updates.

**⚠️ 2. User Behavior Adaptation**

**Goal**: Learn when a user is most/least active and use that pattern.

**Current Status**:

* 🟡 *Partially simulated*. In the simulate\_agent.py or main.py, updates are postponed if user\_active == True or cpu > 80%.
* But this is **rule-based**, not learned over time.

**To achieve**:

* Track user activity over time.
* Build a schedule showing typical idle hours per user/device.
* Use that schedule to **auto-adjust update times** (like personalization).

**⚠️ 3. Risk & Compatibility Assessment**

**Goal**: Evaluate if a patch might cause harm or is unnecessary.

**Current Status**:

* ❌ Not yet implemented in real terms.
* The “High Security Risk” flag is **manually simulated**, not pulled from any real CVE data or compatibility database.

**To achieve**:

* Pull CVE database info.
* Check patch compatibility by simulating whether CPU/battery/network can support it.
* Add a “Patch Safety Verifier” module.

**🟡 4. Dynamic Optimization**

**Goal**: Adapt update parameters dynamically (e.g. schedule, size, batch type) based on real-time device specs.

**Current Status**:

* 🟡 You **check battery/CPU and postpone** if levels are too low or busy.
* But not yet adjusting update **parameters** like:
  + Schedule time
  + Whether to split updates
  + Defer larger patches

**To achieve**:

* Include parameters in update logic like:

python

CopyEdit

if battery < 30 and patch\_size > 100MB:

defer or switch to smaller patch version

**✅ Summary Table**

| **Feature** | **Status** | **What's Done** | **What's Needed** |
| --- | --- | --- | --- |
| Predictive Analytics | ❌ | Logs available | Add ML model using log data |
| User Behavior Adaptation | ⚠️ Partial | Rule-based deferral | Learn from behavior over time |
| Risk/Compatibility Assessment | ❌ | Manual rules | Integrate CVE + patch verifier |
| Dynamic Optimization | ⚠️ Basic | Defers based on battery/cpu | Adjust patch methods/sizes dynamically |

CODE:

# main.py

import time

from explain\_ai import generate\_explanation

from monitor import get\_system\_status

from decision\_engine import decide\_update

from updater import apply\_update, postpone\_update

from logger import log\_decision

print("🚀 Smart Update Manager (Batch Simulation) is starting...\n")

for cycle in range(5):  # Simulate 5 update cycles

    print(f"\n🔄 Cycle {cycle + 1}")

    # 1. Get current system status

    system\_status = get\_system\_status()

    print(f"📊 Current Status: {system\_status}")

    # 2. Decide what to do

    action, reason = decide\_update(system\_status)

    print(f"🤖 Decision: {action} | Reason: {reason}")

    # 3. Try to get AI explanation

    try:

        explanation = generate\_explanation(system\_status, action, reason)

        print(f"🗣️ LLM Explanation: {explanation}")

    except Exception as e:

        explanation = f"⚠️ LLM explanation failed: {e}"

        print(explanation)

    # 4. Log the full decision (system state + action + reason + explanation)

    log\_decision(system\_status, action, reason, explanation)

    # 5. Apply or postpone update

    if action == "Apply":

        apply\_update()

    else:

        postpone\_update()

    print("✅ Action complete.")

    print("-" \* 60)

    time.sleep(2)

# explain\_ai.py

from transformers import pipeline, set\_seed

generator = pipeline("text-generation", model="distilgpt2")

set\_seed(42)

def generate\_explanation(system\_status, action, reason):

    prompt = (

        f"Give a short, polite explanation (with emojis) for why the system decided to {action.lower()} the update. "

        f"The reason is: {reason}.\nExplanation:"

    )

    try:

        output = generator(

            prompt,

            max\_new\_tokens=50,

            do\_sample=True,

            top\_k=50,

            top\_p=0.95,

            temperature=0.7,

            pad\_token\_id=50256,

            truncation=True

        )

        generated = output[0]["generated\_text"]

        explanation = generated.split("Explanation:")[-1].strip()

        return explanation

    except Exception as e:

        return f"⚠️ Hugging Face explanation failed: {e}"

# pages/dashboard.py

import streamlit as st

import pandas as pd

import os

import time

CSV\_FILE = "logs/decisions.csv"

st.set\_page\_config(page\_title="Smart Update Dashboard", layout="wide")

st.title("🧠 Smart Update Manager Dashboard")

# --- 📡 Real-time Monitoring Controls ---

st.sidebar.header("📡 Real-Time Monitoring")

if "monitoring" not in st.session\_state:

    st.session\_state.monitoring = False

refresh\_rate = st.sidebar.slider("⏱️ Refresh Interval (seconds)", 2, 30, 5)

toggle = st.sidebar.button(

    "▶️ Start Monitoring" if not st.session\_state.monitoring else "⏸️ Stop Monitoring"

)

if toggle:

    st.session\_state.monitoring = not st.session\_state.monitoring

# --- Check if log file exists ---

if not os.path.exists(CSV\_FILE):

    st.warning("⚠️ Log file not found. Run the update agent first.")

    st.stop()

# --- Load CSV ---

df = pd.read\_csv(CSV\_FILE, parse\_dates=["timestamp"])

# --- Filters ---

st.sidebar.header("🔍 Filters")

action\_filter = st.sidebar.multiselect("Select Action", df["action"].unique(), default=df["action"].unique())

filtered\_df = df[df["action"].isin(action\_filter)]

# --- Main Table ---

st.subheader("📋 Recent Update Decisions")

st.dataframe(filtered\_df.sort\_values("timestamp", ascending=False), use\_container\_width=True)

# --- Charts ---

col1, col2 = st.columns(2)

with col1:

    st.subheader("📊 Action Count")

    st.bar\_chart(filtered\_df["action"].value\_counts())

with col2:

    st.subheader("🔋 Battery Levels")

    st.line\_chart(filtered\_df[["timestamp", "battery\_level"]].set\_index("timestamp"))

# --- LLM Explanation Preview ---

st.subheader("📄 LLM Explanations (Preview)")

st.write(filtered\_df[["timestamp", "action", "explanation"]].tail(5))

# --- Download Button ---

csv\_download = filtered\_df.to\_csv(index=False).encode("utf-8")

st.download\_button("📥 Download CSV Log", csv\_download, "smart\_update\_log.csv", "text/csv")

# --- 🔁 Rerun Only After Rendering ---

if st.session\_state.monitoring:

    st.sidebar.success("✅ Monitoring enabled...")

    time.sleep(refresh\_rate)

    st.rerun()

else:

    st.sidebar.warning("⏸️ Monitoring paused.")

# pages/simulate\_agent.py

import streamlit as st

import pandas as pd

from datetime import datetime

import os

LOG\_FILE = "logs/decisions.csv"

st.set\_page\_config(page\_title="🧪 Agent Simulation", layout="centered")

st.title("🧪 Smart Update Agent Simulation")

st.markdown("Simulate update scenarios and test the Smart Update logic using manual inputs.")

# Input Fields

st.subheader("🛠️ System Parameters")

col1, col2 = st.columns(2)

with col1:

    battery = st.slider("🔋 Battery Level (%)", 0, 100, 50)

    cpu = st.slider("🧠 CPU Usage (%)", 0, 100, 50)

    risk = st.slider("⚠️ Security Risk (1–10)", 1, 10, 5)

with col2:

    internet = st.selectbox("🌐 Internet Connection", ["online", "offline"])

    user\_active = st.selectbox("🖱️ User Active?", ["Yes", "No"])

# Decision Logic

def decide\_action(battery, cpu, risk, internet, user\_active):

    if internet == "offline":

        return "Postpone", "No internet connection."

    elif battery < 30:

        return "Postpone", "Battery too low."

    elif user\_active == "Yes" or cpu > 80:

        return "Postpone", "System busy or user active."

    elif risk >= 7:

        return "Apply", "High security risk."

    else:

        return "Postpone", "No critical condition met."

# Simulation Button

if st.button("🚀 Simulate Decision"):

    action, reason = decide\_action(battery, cpu, risk, internet, user\_active)

    timestamp = datetime.now()

    # LLM-like explanation (mocked)

    explanation = f"🔍 The system decided to \*\*{action}\*\* the update because: {reason}"

    # Prepare row

    new\_row = pd.DataFrame([{

        "timestamp": timestamp,

        "action": action,

        "reason": reason,

        "explanation": explanation,

        "cpu\_usage": cpu,

        "battery\_level": battery,

        "user\_active": user\_active == "Yes",

        "network": internet,

        "security\_risk\_score": risk

    }])

    # Append to CSV

    if os.path.exists(LOG\_FILE):

        existing = pd.read\_csv(LOG\_FILE)

        updated = pd.concat([existing, new\_row], ignore\_index=True)

    else:

        updated = new\_row

    updated.to\_csv(LOG\_FILE, index=False)

    st.success(f"✅ Simulated decision: \*\*{action}\*\* – {reason}")

    st.code(explanation, language="markdown")

    # Show recent entries

    st.markdown("### 📋 Recent Simulations")

    st.dataframe(updated.tail(5), use\_container\_width=True)

# decision\_engine.py

from config import IDLE\_THRESHOLD, BATTERY\_THRESHOLD, SECURITY\_RISK\_SCORE

def decide\_update(system\_status):

    if system\_status["network"] == "offline":

        return "Postpone", "No internet connection."

    if system\_status["security\_risk\_score"] >= SECURITY\_RISK\_SCORE:

        return "Apply", "High security risk."

    if system\_status["cpu\_usage"] < IDLE\_THRESHOLD and not system\_status["user\_active"]:

        if system\_status["battery\_level"] >= BATTERY\_THRESHOLD:

            return "Apply", "System idle and battery sufficient."

        else:

            return "Postpone", "Battery too low."

    else:

        return "Postpone", "System busy or user active."

# logger.py

import os

import csv

from datetime import datetime

# Ensure log directories exist

LOG\_DIR = "logs"

os.makedirs(LOG\_DIR, exist\_ok=True)

# File paths

TEXT\_LOG\_FILE = os.path.join(LOG\_DIR, "decisions.log")

CSV\_LOG\_FILE = os.path.join(LOG\_DIR, "decisions.csv")

# Write CSV header only once (if file doesn't exist)

if not os.path.exists(CSV\_LOG\_FILE):

    with open(CSV\_LOG\_FILE, "w", newline='', encoding="utf-8") as f:

        writer = csv.writer(f)

        writer.writerow([

            "timestamp", "action", "raw\_reason", "llm\_explanation",

            "cpu\_usage", "battery\_level", "user\_active", "network", "security\_risk\_score"

        ])

def log\_decision(system\_status, action, raw\_reason, llm\_explanation):

    timestamp = datetime.now()

    # 1️⃣ Text Log (Developer Friendly)

    with open(TEXT\_LOG\_FILE, "a", encoding="utf-8") as log:

        log.write(

            f"{timestamp} | Action: {action} | Raw Reason: {raw\_reason} | "

            f"LLM Explanation: {llm\_explanation} | Status: {system\_status}\n"

        )

    # 2️⃣ CSV Log (Data Friendly)

    with open(CSV\_LOG\_FILE, "a", newline='', encoding="utf-8") as f:

        writer = csv.writer(f)

        writer.writerow([

            timestamp,

            action,

            raw\_reason,

            llm\_explanation,

            system\_status["cpu\_usage"],

            system\_status["battery\_level"],

            system\_status["user\_active"],

            system\_status["network"],

            system\_status["security\_risk\_score"]

        ])