

3. Supplement Advisory Backend and ML Model

Personalized supplement effectiveness prediction, safety rules, product matching, and Random Forest inference

Viva focus: explain how a supplement request becomes a ranked, safety-screened recommendation and how the result is stored for the authenticated user.

1. Module overview

The supplement module combines a trained Random Forest effectiveness model with deterministic safety and product rules. The Express service accepts a user's goal, body profile, diet, budget, medication list, allergies, and conditions. The controller passes the normalized request to a Python inference boundary. Python predicts effectiveness for goal-relevant supplements, checks interactions and exclusions, matches affordable products, ranks the results, and returns one explainable JSON response.

2. Request and data flow

POST /api/supplement/advise -> optional auth middleware -> controller copies the request contract -> child_process.spawn runs ml_service/supplement_main.py -> Python loads the persisted encoders and Random Forest -> goal candidates are predicted and screened against medication, allergen, diet, condition, and budget data -> recommendations are sorted by score -> Node stores SupplementAdviceLog when a valid user and MongoDB are available -> JSON returns to the client.

A useful viva sentence:

“The route accepts both public and personalized advice, the controller coordinates the Python process and persistence, and the Python layer combines learned effectiveness with deterministic safety rules before returning a ranked JSON contract.”

3. API surface

Method	Path	Purpose
POST	/api/supplement/advise	Generate personalized, safety-screened supplement recommendations.
GET	/api/supplement/history	Read the authenticated user's advice history.
DELETE	/api/supplement/history	Clear the authenticated user's advice history.
GET	/api/supplement/saved	Read the user's saved supplement list.
PUT	/api/supplement/saved	Replace the user's saved supplement list.
GET	/api/supplement/progress	Read supplement usage and progress notes.
PUT	/api/supplement/progress	Create or update a progress note.
DELETE	/api/supplement/progress/:id	Delete one progress note.

4. Important concepts to explain

- The Random Forest predicts effectiveness as Low, Medium, or High from encoded profile and supplement features; it does not replace the safety rule engine.
- The same LabelEncoder objects fitted during training are persisted and reused during inference so categorical values keep the same numerical meaning.
- Goal aliases and profile values are normalized at the boundary. Age and height become model buckets, while weight, body fat, and budget are validated before prediction.

- Safety screening is deterministic and explainable: medication interactions, allergen catalog matches, vegan/vegetarian restrictions, and condition rules can exclude a recommendation.
- Product matching is independent of the ML prediction. It searches the supplement product catalog, converts prices to LKR, enforces the selected maximum budget, de-duplicates products, and keeps the top three.
- A hard exclusion becomes Not Recommended with score 0. Otherwise the score combines effectiveness, product availability, and warning severity into Recommended, Caution, or Not Recommended.
- The advice endpoint uses optional authentication: anonymous users can receive advice, while authenticated users also get a MongoDB history record.

5. Full source code

The following listings are copied from the current supplement implementation. Comments are retained. Long lines are wrapped visually only so they fit on paper; the code content is not intentionally omitted.

ress/services/supplement-service/server.js

How this code works

- Builds the Supplement Advisory Express service and mounts all supplement routes.
- Initializes MongoDB when MONGO_URI is configured and reports the persistence mode.
- Keeps HTTP orchestration separate from the Python model and rule engine.

```
const express = require("express");
const cors = require("cors");
const dns = require("dns");
const dotenv = require("dotenv");
const mongoose = require("mongoose");

dns.setServers(["8.8.8.8", "1.1.1.1"]);

dotenv.config();

const supplementRoutes = require("../src/routes/supplementRoutes");

const app = express();

app.use(cors());
app.use(express.json());

app.use("/api/supplement", supplementRoutes);

app.get("/", (req, res) => {
  res.json({ message: "Supplement Advisory Service is running" });
});

const PORT = process.env.PORT || 5006;

app.listen(PORT, () => {
  console.log(`Supplement Service running on port ${PORT}`);
});

if (process.env.MONGO_URI) {
  mongoose
    .connect(process.env.MONGO_URI, {
      serverSelectionTimeoutMS: 10000,
      socketTimeoutMS: 45000,
    })
    .then(() => console.log("Supplement Service MongoDB connected"))
    .catch((err) => console.error("MongoDB connection error (supplement):", err));
} else {
  console.log("Supplement Service running without MongoDB (MONGO_URI not set)");
}
```

ress/services/supplement-service/src/routes/supplementRoutes.js

How this code works

- Defines advice, history, saved-supplement, and progress-note endpoints.
- Uses optional authentication for advice and required authentication for user-owned records.

```
const express = require("express");
const router = express.Router();

const authMiddleware = require("../middleware/authMiddleware");
const { optionalAuthMiddleware } = authMiddleware;
const {
  getSupplementAdvice,
  getMySupplementHistory,
  clearMySupplementHistory,
  getMySavedSupplements,
}
```

```

    saveMySupplements,
    getMyProgressNotes,
    saveMyProgressNote,
    deleteMyProgressNote,
  } = require("../controllers/supplementController");

  router.post("/advise", optionalAuthMiddleware, getSupplementAdvice);
  router.get("/history", authMiddleware, getMySupplementHistory);
  router.delete("/history", authMiddleware, clearMySupplementHistory);
  router.get("/saved", authMiddleware, getMySavedSupplements);
  router.put("/saved", authMiddleware, saveMySupplements);
  router.get("/progress", authMiddleware, getMyProgressNotes);
  router.put("/progress", authMiddleware, saveMyProgressNote);
  router.delete("/progress/:id", authMiddleware, deleteMyProgressNote);

  module.exports = router;

```

ress/services/supplement-service/src/middleware/authMiddleware.js

How this code works

- Verifies Bearer JWTs and attaches userId and email to req.user.
- The optional variant allows public advice while still enabling per-user logging.

```

const jwt = require("jsonwebtoken");

function authMiddleware(req, res, next) {
  try {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
      return res.status(401).json({
        success: false,
        message: "No token provided",
      });
    }

    const token = authHeader.split(" ")[1];

    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    req.user = {
      userId: decoded.userId,
      email: decoded.email,
    };

    next();
  } catch (error) {
    return res.status(401).json({
      success: false,
      message: "Invalid or expired token",
    });
  }
}

function optionalAuthMiddleware(req, res, next) {
  try {
    const authHeader = req.headers.authorization;
    if (authHeader && authHeader.startsWith("Bearer ")) {
      const token = authHeader.split(" ")[1];
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      req.user = {
        userId: decoded.userId,
        email: decoded.email,
      };
    }
  } catch (error) {
    // Invalid or expired tokens are treated as unauthenticated here.
  }
  next();
}

module.exports = authMiddleware;
module.exports.optionalAuthMiddleware = optionalAuthMiddleware;

```

ress/services/supplement-service/src/controllers/supplementController.js

How this code works

- Builds the canonical Python input, finds Python on the host, spawns the ML process, and parses JSON stdout.
- Returns explicit errors for missing Python, invalid output, and non-zero ML exits.
- Persists advice, saved supplements, and progress notes through MongoDB models.

```
const { spawn, execSync } = require("child_process");
const fs = require("fs");
const os = require("os");
const path = require("path");
const SupplementAdviceLog = require("../models/SupplementAdviceLog");
const SavedSupplement = require("../models/SavedSupplement");
const SupplementProgressNote = require("../models/SupplementProgressNote");

exports.getSupplementAdvice = async (req, res) => {
  try {
    const inputData = {
      goal: req.body.goal,
      age: req.body.age,
      weight: req.body.weight,
      height: req.body.height,
      gender: req.body.gender,
      diet: req.body.diet,
      budget: req.body.budget,
      medications: req.body.medications,
      allergies: req.body.allergies,
      conditions: req.body.conditions,
    };

    const pythonScriptPath = path.join(
      __dirname,
      "..",
      "..",
      "..",
      "ml_service",
      "supplement_main.py"
    );

    const defaultVenvPath = path.join(
      __dirname,
      "..",
      "..",
      "..",
      "venv",
      "Scripts",
      "python.exe"
    );

    const homeVenvPath = path.join(os.homedir(), "venv", "Scripts", "python.exe");
    const downloadsEmbedPath = path.join(
      os.homedir(),
      "Downloads",
      "python-3.11.0-embed-amd64",
      "python.exe"
    );

    const fileCandidates = [
      process.env.PYTHON_PATH,
      defaultVenvPath,
      homeVenvPath,
      downloadsEmbedPath,
    ].filter(Boolean);

    const commandCandidates = ["python3", "python"];

    let pythonPath = fileCandidates.find((candidate) => {
      try {
        return fs.existsSync(candidate) && fs.lstatSync(candidate).isFile();
      } catch {
        return false;
      }
    });

    if (!pythonPath) {
```

```

pythonPath = commandCandidates.find((cmd) => {
  try {
    execSync(`which ${cmd}`, { stdio: "ignore" });
    return true;
  } catch {
    return false;
  }
});
}

if (!pythonPath) {
  return res.status(500).json({
    success: false,
    message:
      "Python executable not found. Set PYTHON_PATH in .env or install Python on PATH.",
    checkedPaths: [...fileCandidates, ...commandCandidates],
  });
}

const pythonProcess = spawn(pythonPath, [
  pythonScriptPath,
  JSON.stringify(inputData),
]);

let result = "";
let error = "";

pythonProcess.stdout.on("data", (data) => {
  result += data.toString();
});

pythonProcess.stderr.on("data", (data) => {
  error += data.toString();
});

pythonProcess.on("error", (spawnError) => {
  return res.status(500).json({
    success: false,
    message: "Failed to start Python process",
    error: spawnError.message,
  });
});

pythonProcess.on("close", async (code) => {
  let parsedResult;

  try {
    parsedResult = JSON.parse(result);
  } catch (parseError) {
    return res.status(500).json({
      success: false,
      message: "Failed to parse Python output",
      rawOutput: result,
      pythonError: error,
    });
  }

  if (code !== 0) {
    return res.status(500).json({
      success: false,
      message: "Python ML service failed",
      error,
      result: parsedResult,
    });
  }

  if (parsedResult.success && req.user) {
    try {
      await SupplementAdviceLog.create({
        userId: req.user.userId,
        goal: parsedResult.goal,
        diet: parsedResult.diet,
        budget: parsedResult.budget,
        medications: inputData.medications,
        allergies: inputData.allergies,
        conditions: inputData.conditions,
        recommendations: parsedResult.recommendations,
        excludedCount: parsedResult.excluded_count,
      });
    } catch (dbErr) {
      console.warn("Supplement log skipped (no DB):", dbErr.message);
    }
  }
}

```

```

    }

    return res.json(parsedResult);
  });
} catch (error) {
  return res.status(500).json({
    success: false,
    message: "Supplement advice generation failed",
    error: error.message,
  });
}
};

exports.getMySupplementHistory = async (req, res) => {
  try {
    const logs = await SupplementAdviceLog.find({ userId: req.user.userId }).sort({
      createdAt: -1,
    });

    return res.json({
      success: true,
      count: logs.length,
      data: logs,
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      message: "Failed to fetch supplement history",
      error: error.message,
    });
  }
};

exports.clearMySupplementHistory = async (req, res) => {
  try {
    const result = await SupplementAdviceLog.deleteMany({ userId: req.user.userId });

    return res.json({
      success: true,
      message: "Supplement history cleared",
      deletedCount: result.deletedCount,
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      message: "Failed to clear supplement history",
      error: error.message,
    });
  }
};

exports.getMySavedSupplements = async (req, res) => {
  try {
    const saved = await SavedSupplement.findOne({ userId: req.user.userId }).lean();

    return res.json({
      success: true,
      data: saved ? saved.supplements : [],
      hasRecord: Boolean(saved),
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      message: "Failed to fetch saved supplements",
      error: error.message,
    });
  }
};

exports.saveMySupplements = async (req, res) => {
  try {
    if (!Array.isArray(req.body.supplements)) {
      return res.status(400).json({
        success: false,
        message: "supplements must be an array",
      });
    }

    const saved = await SavedSupplement.findOneAndUpdate(
      { userId: req.user.userId },
      { $set: { supplements: req.body.supplements } },
      { new: true, upsert: true, setDefaultsOnInsert: true, runValidators: true }
    );
  }
};

```

```

    ).lean();

    return res.json({
      success: true,
      data: saved.supplements,
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      message: "Failed to save supplements",
      error: error.message,
    });
  }
};

exports.getMyProgressNotes = async (req, res) => {
  try {
    const notes = await SupplementProgressNote.find({
      userId: req.user.userId,
    })
      .sort({ savedAt: -1, createdAt: -1 })
      .lean();

    return res.json({
      success: true,
      count: notes.length,
      data: notes.map((note) => ({
        id: note.clientId,
        supplement: note.supplement,
        date: note.date,
        time: note.time,
        used: note.used,
        reason: note.reason || "",
        notes: note.notes || "",
        savedAt: note.savedAt || note.createdAt,
      })),
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      message: "Failed to fetch supplement progress notes",
      error: error.message,
    });
  }
};

exports.saveMyProgressNote = async (req, res) => {
  try {
    const { id, supplement, date, time, used, reason, notes, savedAt } =
      req.body || {};

    if (!id || !supplement || !date || !time || !["yes", "no"].includes(used)) {
      return res.status(400).json({
        success: false,
        message: "id, supplement, date, time, and a valid used value are required",
      });
    }

    if (used === "no" && !String(reason || "").trim()) {
      return res.status(400).json({
        success: false,
        message: "A reason is required when the supplement was not used",
      });
    }

    const saved = await SupplementProgressNote.findOneAndUpdate(
      { userId: req.user.userId, clientId: String(id) },
      {
        $set: {
          supplement: String(supplement).trim(),
          date: String(date),
          time: String(time),
          used,
          reason: String(reason || ""),
          notes: String(notes || ""),
          savedAt: savedAt ? new Date(savedAt) : new Date(),
        },
        $setOnInsert: {
          userId: req.user.userId,
          clientId: String(id),
        },
      },
    );
  }
};

```

```

    { new: true, upsert: true, setDefaultsOnInsert: true, runValidators: true }
  ).lean();

  return res.json({
    success: true,
    data: {
      id: saved.clientId,
      supplement: saved.supplement,
      date: saved.date,
      time: saved.time,
      used: saved.used,
      reason: saved.reason || "",
      notes: saved.notes || "",
      savedAt: saved.savedAt || saved.createdAt,
    },
  });
} catch (error) {
  return res.status(500).json({
    success: false,
    message: "Failed to save supplement progress note",
    error: error.message,
  });
}
};

exports.deleteMyProgressNote = async (req, res) => {
  try {
    const deleted = await SupplementProgressNote.findOneAndDelete({
      userId: req.user.userId,
      clientId: String(req.params.id || ""),
    });

    if (!deleted) {
      return res.status(404).json({
        success: false,
        message: "Progress note not found",
      });
    }

    return res.json({
      success: true,
      message: "Progress note deleted",
      id: deleted.clientId,
    });
  } catch (error) {
    return res.status(500).json({
      success: false,
      message: "Failed to delete supplement progress note",
      error: error.message,
    });
  }
};
};

```

ress/services/supplement-service/src/models/SupplementAdviceLog.js

How this code works

- Stores the authenticated user's input context and generated recommendation array.
- Timestamps make history ordering deterministic.

```

const mongoose = require("mongoose");

const supplementAdviceLogSchema = new mongoose.Schema(
  {
    userId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      index: true,
    },
    goal: String,
    diet: String,
    budget: String,
    medications: String,
    allergies: String,
    conditions: String,
    recommendations: Array,
    excludedCount: Number,
  },
  { timestamps: true }
);

```

```
);

module.exports = mongoose.model("SupplementAdviceLog", supplementAdviceLogSchema);
```

ress/services/supplement-service/src/models/SavedSupplement.js

How this code works

- Stores one saved supplement document per user with a flexible recommendation payload.

```
const mongoose = require("mongoose");

const savedSupplementSchema = new mongoose.Schema(
  {
    userId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      unique: true,
      index: true,
    },
    supplements: {
      type: [mongoose.Schema.Types.Mixed],
      default: [],
    },
  },
  { timestamps: true }
);

module.exports = mongoose.model("SavedSupplement", savedSupplementSchema);
```

ress/services/supplement-service/src/models/SupplementProgressNote.js

How this code works

- Stores usage status, reason, notes, date, time, and a client id.
- The compound userId/clientId index makes updates idempotent per user.

```
const mongoose = require("mongoose");

const supplementProgressNoteSchema = new mongoose.Schema(
  {
    userId: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      index: true,
    },
    clientId: {
      type: String,
      required: true,
    },
    supplement: {
      type: String,
      required: true,
      trim: true,
    },
    date: {
      type: String,
      required: true,
    },
    time: {
      type: String,
      required: true,
    },
    used: {
      type: String,
      enum: ["yes", "no"],
      required: true,
    },
    reason: {
      type: String,
      default: "",
    },
    notes: {
      type: String,
      default: "",
    },
    savedAt: {
```

```

        type: Date,
        default: Date.now,
    },
},
{ timestamps: true }
);

supplementProgressNoteSchema.index({ userId: 1, clientId: 1 }, { unique: true });

module.exports = mongoose.model(
    "SupplementProgressNote",
    supplementProgressNoteSchema
);

```

ress/ml_service/supplement_main.py

How this code works

- Loads the Random Forest, encoders, and four safety/product catalogs.
- Combines prediction, deterministic warnings, budget filtering, scoring, and response shaping.
- Prints exactly one JSON result for the Node controller to parse.

```

import sys
import json
import re
import pandas as pd
import joblib
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent

MODEL_PATH = BASE_DIR / "models" / "supplement_effectiveness_model.pkl"
ENCODER_PATH = BASE_DIR / "models" / "supplement_encoders.pkl"

FITNESS_PATH = BASE_DIR / "datasets" / "fitness_supplements_dataset.csv"
INTERACTIONS_PATH = BASE_DIR / "datasets" / "supplement_drug_interactions.csv"
ALLERGENS_PATH = BASE_DIR / "datasets" / "supplement_allergens.csv"
PRODUCTS_PATH = BASE_DIR / "datasets" / "supplement_products.csv"
ALLERGY_ALIASES_PATH = BASE_DIR / "datasets" / "food_allergy_aliases.csv"

model = joblib.load(MODEL_PATH)
encoders = joblib.load(ENCODER_PATH)

fitness_df = pd.read_csv(FITNESS_PATH)
interactions_df = pd.read_csv(INTERACTIONS_PATH, engine="python", on_bad_lines="skip")
allergens_df = pd.read_csv(ALLERGENS_PATH, engine="python", on_bad_lines="skip")
products_df = pd.read_csv(PRODUCTS_PATH, engine="python", on_bad_lines="skip")
allergy_aliases_df = pd.read_csv(ALLERGY_ALIASES_PATH, engine="python", on_bad_lines="skip")

CATEGORICAL_COLS = [
    "Gender",
    "Age",
    "Height(cm)",
    "Fitness_Level",
    "Weekly_Training",
    "Training_Type",
    "Supplement",
    "Supplement_Type",
    "Diet_Type",
]
NUMERIC_COLS = ["Weight(kg)", "Body_Fat(%)"]
FEATURES = CATEGORICAL_COLS + NUMERIC_COLS

EFFECTIVENESS_LABEL = {0: "Low", 1: "Medium", 2: "High"}

SUPPLEMENTS = sorted(fitness_df["Supplement"].dropna().unique().tolist())
SUPPLEMENT_TYPE_MAP = (
    fitness_df.groupby("Supplement")["Supplement_Type"]
    .agg(lambda x: x.value_counts().idxmax())
    .to_dict()
)

SUPPLEMENT_ICON = {
    "BCAA": "■",
    "Creatine": "■",
    "Energy_Drink": "■",
    "Fish_Oil": "■",

```

```

    "L_Carnitine": "■",
    "Multivitamin": "■",
    "Pre_Workout": "■",
    "Whey_Protein": "■",
}

GOAL_TO_TRAINING_TYPE = {
    "Weight Loss": "Endurance",
    "Muscle Gain": "Strength_Training",
    "General Fitness": "Strength_Training",
    "Improve Immunity": "Flexibility",
    "Increase Energy": "HIIT",
    "Bone Health": "Flexibility",
}

GOAL_PREFERRED_SUPPLEMENTS = {
    "Weight Loss": ["L_Carnitine", "Fish_Oil", "Multivitamin", "BCAA"],
    "Muscle Gain": ["Whey_Protein", "Creatine", "BCAA", "Multivitamin"],
    "General Fitness": ["Multivitamin", "Fish_Oil", "BCAA", "Whey_Protein"],
    "Improve Immunity": ["Multivitamin", "Fish_Oil", "BCAA"],
    "Increase Energy": ["Energy_Drink", "Pre_Workout", "Multivitamin", "BCAA"],
    "Bone Health": ["Multivitamin", "Fish_Oil", "BCAA"],
}

DIET_RESTRICTIONS = {
    "Vegan": {
        "Fish_Oil": ("high", "Fish oil is animal-derived.", "Choose an algae-based omega-3 instead."),
        "Whey_Protein": ("high", "Whey protein is dairy-derived.", "Choose a plant-based protein instead."),
    },
    "Vegetarian": {
        "Fish_Oil": ("high", "Fish oil is not vegetarian.", "Choose an algae-based omega-3 instead."),
    },
}

CONDITION_RULES = [
    ({ "high blood pressure", "hypertension", "high bp", "blood pressure"},
    { "Energy_Drink", "Pre_Workout"}, "high",
    "This stimulant product may raise blood pressure.",
    "Avoid unless a clinician approves it."),
    ({ "diabetes", "diabetic", "high blood sugar", "type 1 diabetes", "type 2 diabetes"},
    { "Energy_Drink"}, "moderate",
    "Energy drinks may affect blood-glucose management.",
    "Check the label and confirm suitability with a clinician."),
    ({ "kidney disease", "kidney problem", "renal disease", "kidney failure"},
    { "Creatine", "Whey_Protein", "BCAA"}, "high",
    "This product may be unsuitable with kidney disease.",
    "Do not use without clinician guidance."),
    ({ "liver disease", "liver problem", "liver failure", "hepatitis"},
    { "Energy_Drink", "Pre_Workout"}, "moderate",
    "Concentrated stimulant products may add avoidable risk with liver disease.",
    "Confirm suitability with a clinician before use."),
    ({ "bleeding disorder", "bleeding problem", "hemophilia"},
    { "Fish_Oil"}, "high",
    "Fish oil can increase bleeding risk.",
    "Avoid unless supervised by a clinician."),
]

DIET_TO_DIET_TYPE = {
    "Vegan": "Vegan",
    "Vegetarian": "Regular",
    "Non-Vegetarian": "High_Protein",
}

PRODUCT_KEYWORDS = {
    "BCAA": ["bcaa", "amino"],
    "Creatine": ["creatine"],
    "Energy_Drink": ["energy", "caffeine", "guarana"],
    "Fish_Oil": ["fish oil", "omega"],
    "L_Carnitine": ["carnitine"],
    "Multivitamin": ["multivitamin"],
    "Pre_Workout": ["pre workout", "pre-workout", "workout", "citrulline"],
    "Whey_Protein": ["whey", "protein"],
}

BUDGET_MAX = {
    "Under Rs.1000": 1000,
    "Rs.1000-3000": 3000,
    "Rs.1000\u20133000": 3000,
    "No Limit": float("inf"),
}

GOAL_ALIASES = {

```

```

        "Energy Boost": "Increase Energy",
        "Immunity": "Improve Immunity",
    }
    SUPPORTED_GENDERS = {"Male", "Female"}
    SUPPORTED_DIETS = set(DIET_TO_DIET_TYPE)
    SUPPORTED_FITNESS_LEVELS = {"Beginner", "Intermediate", "Advanced"}
    SUPPORTED_WEEKLY_TRAINING = {"Less_than_3", "3-5_times", "More_than_5"}
    USD_TO_LKR = 300

def normalize(text):
    return re.sub(r"[^a-z0-9\s]", " ", str(text).lower()).strip()

def bucket_age(age):
    try:
        age = int(age)
    except (TypeError, ValueError):
        return "25-35"
    if age < 25:
        return "Under_25"
    if age <= 35:
        return "25-35"
    return "Over_35"

def bucket_height(height):
    try:
        height = float(height)
    except (TypeError, ValueError):
        return "170-180"
    if height < 160:
        return "150-160"
    if height < 170:
        return "160-170"
    if height < 180:
        return "170-180"
    if height < 190:
        return "180-190"
    return "Above_190"

def encode_row(row_dict):
    df = pd.DataFrame([row_dict])
    for col, le in encoders.items():
        df[col] = df[col].astype(str)
        known = set(le.classes_)
        unknown = [x for x in df[col].tolist() if x not in known]
        if unknown:
            raise ValueError(f"Unsupported value for {col}: {unknown[0]}")
        df[col] = le.transform(df[col])
    return df[FEATURES]

def predict_effectiveness(user, supplement):
    row = {
        "Gender": user["gender"],
        "Age": bucket_age(user["age"]),
        "Height(cm)": bucket_height(user["height"]),
        "Fitness_Level": user["fitness_level"],
        "Weekly_Training": user["weekly_training"],
        "Training_Type": user["training_type"],
        "Supplement": supplement,
        "Supplement_Type": SUPPLEMENT_TYPE_MAP.get(supplement, "Standard"),
        "Diet_Type": user["diet_type"],
        "Weight(kg)": user["weight"],
        "Body_Fat(%)": user["body_fat"],
    }
    encoded = encode_row(row)
    pred = int(model.predict(encoded)[0])
    proba = model.predict_proba(encoded)[0].tolist()
    return pred, proba

def check_drug_interactions(supplement, medications_text):
    if not medications_text:
        return []
    supplement_norm = normalize(supplement.replace("_", " "))
    meds = [normalize(m) for m in re.split(r"[;/]", medications_text) if normalize(m)]
    if not meds:
        return []
    warnings = []

```

```

for _, row in interactions_df.iterrows():
    sup_norm = normalize(row.get("Supplement", ""))
    if not sup_norm:
        continue
    if supplement_norm not in sup_norm and sup_norm not in supplement_norm:
        continue
    drug_norm = normalize(row.get("Drug", ""))
    for med in meds:
        med_words = set(med.split())
        drug_words = set(drug_norm.split())
        if med in drug_norm or drug_norm in med or bool(med_words & drug_words):
            warnings.append({
                "supplement": supplement.replace("_", " "),
                "drug": row.get("Drug", ""),
                "severity": row.get("Severity", "Moderate"),
                "interaction": row.get("Interaction", ""),
                "recommendation": row.get("Recommendation", ""),
            })
    return warnings

def check_diet_compatibility(supplement, diet):
    restriction = DIET_RESTRICTIONS.get(diet, {}).get(supplement)
    if not restriction:
        return []
    severity, reason, recommendation = restriction
    return [{
        "supplement": supplement.replace("_", " "),
        "severity": severity,
        "reason": reason,
        "recommendation": recommendation,
    }]

def check_condition_safety(supplement, conditions_text):
    if not conditions_text or normalize(conditions_text) in {"none", "no", "n a", "na"}:
        return []
    condition_terms = {
        normalize(term)
        for term in re.split(r"[; /]", conditions_text)
        if normalize(term)
    }
    warnings = []
    for terms, supplements, severity, reason, recommendation in CONDITION_RULES:
        matched_condition = next(
            (
                condition
                for condition in condition_terms
                if any(term in condition or condition in term for term in terms)
            ),
            None,
        )
        if matched_condition and supplement in supplements:
            warnings.append({
                "supplement": supplement.replace("_", " "),
                "condition": matched_condition,
                "severity": severity,
                "reason": reason,
                "recommendation": recommendation,
            })
    return warnings

def _allergy_alias_groups():
    groups = {}
    for _, row in allergy_aliases_df.iterrows():
        canonical = normalize(row.get("Canonical_Allergen", ""))
        if not canonical:
            continue
        aliases = [normalize(x) for x in str(row.get("Aliases", "")).split(";") if normalize(x)]
        groups[canonical] = set([canonical] + aliases)
    return groups

ALLERGY_ALIAS_GROUPS = _allergy_alias_groups()

def expand_allergy_terms(allergies_text):
    """Normalize free-text allergy input into user terms + canonical allergen groups.

    This is deterministic safety screening, separate from ML prediction.
    For conservative screening, user input such as 'seafood' maps to both Fish

```

```

and Crustacean shellfish because both alias groups contain that term.
"""
raw_items = [normalize(a) for a in re.split(r"[;/]", allergies_text or "") if normalize(a)]
expanded = set(raw_items)
canonical_matches = set()
for raw in raw_items:
    for canonical, aliases in ALLERGY_ALIAS_GROUPS.items():
        if any(alias and (alias in raw or raw in alias) for alias in aliases):
            canonical_matches.add(canonical)
            expanded.add(canonical)
            expanded.update(aliases)
return expanded, canonical_matches

def check_allergens(supplement, allergies_text):
    if not allergies_text:
        return []
    supplement_norm = normalize(supplement.replace("_", " "))
    allergy_terms, canonical_matches = expand_allergy_terms(allergies_text)
    if not allergy_terms:
        return []

    warnings = []
    seen = set()
    for _, row in allergens_df.iterrows():
        sup_norm = normalize(row.get("Supplement", ""))
        if not sup_norm:
            continue
        if not (supplement_norm in sup_norm or sup_norm in supplement_norm):
            continue

        searchable = " ".join([
            normalize(row.get("Allergen/Sensitivity", "")),
            normalize(row.get("Allergen_Category", "")),
            normalize(row.get("User_Input_Aliases", "")),
        ])
        row_catalog = {normalize(x) for x in str(row.get("Alias_Catalog", "")).split(";") if
            normalize(x)}
        matched = any(term and term in searchable for term in allergy_terms) or bool(row_catalog &
            canonical_matches)
        if not matched:
            continue

        key = (supplement_norm, normalize(row.get("Allergen/Sensitivity", "")))
        if key in seen:
            continue
        seen.add(key)
        warnings.append({
            "supplement": supplement.replace("_", " "),
            "allergen": row.get("Allergen/Sensitivity", ""),
            "allergen_category": row.get("Allergen_Category", ""),
            "severity": row.get("Severity", "Moderate"),
            "reaction": row.get("Reaction", ""),
            "recommendation": row.get("Recommendation", ""),
            "source_dependent": "source dependent" in normalize(row.get("Allergen/Sensitivity", "")),
        })
    return warnings

def find_matching_products(supplement, max_budget):
    keywords = PRODUCT_KEYWORDS.get(supplement, [normalize(supplement.replace("_", " "))])
    df = products_df.copy()
    df["_cat_norm"] = df["product_category"].apply(normalize)
    df["_name_norm"] = df["product_name"].apply(normalize)

    mask = df["_cat_norm"].apply(lambda c: any(k in c for k in keywords)) | \
        df["_name_norm"].apply(lambda n: any(k in n for k in keywords))
    matched = df[mask].copy()
    if matched.empty:
        return []

    matched["price_lkr"] = matched["price"].fillna(0) * USD_TO_LKR
    if max_budget != float("inf"):
        # Never fall back to over-budget products. An empty result is safer
        # than displaying a product the user explicitly excluded by price.
        matched = matched[matched["price_lkr"] <= max_budget]
        if matched.empty:
            return []

    # Supplement-only product de-duplication. Keep the highest-rated row when
    # the same brand/product appears more than once in the product dataset.
    # This operates only on the local matched-product result and does not

```

```

# modify the shared CSV or any diet/exercise/habit function.
matched["_brand_norm"] = matched["brand_name"].fillna("").apply(normalize)
matched["_product_norm"] = matched["product_name"].fillna("").apply(normalize)
matched = matched.sort_values(by="overall_rating", ascending=False)
matched = matched.drop_duplicates(subset=["_brand_norm", "_product_norm"], keep="first")
matched = matched.head(3)

results = []
for _, row in matched.iterrows():
    results.append({
        "brand": row.get("brand_name", "Unknown"),
        "product_name": row.get("product_name", ""),
        "price_lkr": round(float(row.get("price_lkr", 0)), 2),
        "rating": row.get("overall_rating", None),
    })
return results

def score_and_tier(
    effectiveness_level,
    drug_warnings,
    allergen_warnings,
    diet_warnings,
    condition_warnings,
    has_product_in_budget,
):
    score = 30

    if effectiveness_level == "High":
        score += 30
    elif effectiveness_level == "Medium":
        score += 15

    if has_product_in_budget:
        score += 10

    hard_excluded = False
    for w in drug_warnings:
        sev = str(w.get("severity", "")).lower()
        if sev == "high":
            hard_excluded = True
            score -= 80
        elif sev == "moderate":
            score -= 40
        else:
            score -= 15

    for w in allergen_warnings + diet_warnings + condition_warnings:
        sev = str(w.get("severity", "")).lower()
        source_dependent = bool(w.get("source_dependent", False))
        # A confirmed high-severity ingredient match is a hard exclusion.
        # Source-dependent rows (for example some BCAA/protein products) are
        # downgraded to caution because the actual product label determines
        # whether that allergen source is present.
        if sev == "high" and not source_dependent:
            hard_excluded = True
        elif sev in {"high", "moderate"}:
            score -= 30
        else:
            score -= 10

    score = max(0, min(100, score))

    if hard_excluded:
        tier = "Not Recommended"
        score = 0
    elif score >= 70:
        tier = "Recommended"
    elif score >= 30:
        tier = "Caution"
    else:
        tier = "Not Recommended"

    return score, tier, hard_excluded

def _number(value, name, default=None, min_value=None, max_value=None):
    if value is None or value == "":
        if default is not None:
            return default
        raise ValueError(f"{name} is required")
    try:

```

```

        number = float(value)
    except (TypeError, ValueError):
        raise ValueError(f"{name} must be numeric")
    if min_value is not None and number < min_value:
        raise ValueError(f"{name} must be at least {min_value}")
    if max_value is not None and number > max_value:
        raise ValueError(f"{name} must be at most {max_value}")
    return number

def advise_supplements(request_data):
    # Backward-compatible aliases are accepted, but every value is normalized to
    # one canonical contract before the model is called.
    raw_goal = request_data.get("goal", "General Fitness")
    goal = GOAL_ALIASES.get(raw_goal, raw_goal)
    diet = request_data.get("diet") or request_data.get("dietType") or "Non-Vegetarian"
    budget = request_data.get("budget") or "No Limit"
    medications = request_data.get("medications", "") or request_data.get("meds", "")
    allergies = request_data.get("allergies", "") or ""
    conditions = request_data.get("conditions", "") or request_data.get("medicalConditions", "") or ""
    gender = request_data.get("gender", "")
    fitness_level = request_data.get("fitness_level") or "Intermediate"
    weekly_training = request_data.get("weekly_training") or "3-5_times"

    if goal not in GOAL_TO_TRAINING_TYPE or goal not in GOAL_PREFERRED_SUPPLEMENTS:
        raise ValueError(
            "Unsupported goal. Use one of: " + ", ".join(GOAL_PREFERRED_SUPPLEMENTS.keys())
        )
    if gender not in SUPPORTED_GENDERS:
        raise ValueError("Unsupported gender for this trained model. Use Male or Female.")
    if diet not in SUPPORTED_DIETS:
        raise ValueError("Unsupported diet. Use Vegan, Vegetarian, or Non-Vegetarian.")
    if budget not in BUDGET_MAX:
        raise ValueError("Unsupported budget. Use Under Rs.1000, Rs.1000-3000, or No Limit.")
    if fitness_level not in SUPPORTED_FITNESS_LEVELS:
        raise ValueError("Unsupported fitness_level. Use Beginner, Intermediate, or Advanced.")
    if weekly_training not in SUPPORTED_WEEKLY_TRAINING:
        raise ValueError("Unsupported weekly_training. Use Less_than_3, 3-5_times, or More_than_5.")

    age = _number(request_data.get("age"), "age", min_value=13, max_value=100)
    height = _number(request_data.get("height"), "height", min_value=100, max_value=250)
    weight = _number(request_data.get("weight"), "weight", min_value=25, max_value=300)
    body_fat = _number(request_data.get("body_fat"), "body_fat", default=20, min_value=2, max_value=70)

    user = {
        "gender": gender,
        "age": age,
        "height": height,
        "weight": weight,
        "body_fat": body_fat,
        "fitness_level": fitness_level,
        "weekly_training": weekly_training,
        "training_type": GOAL_TO_TRAINING_TYPE[goal],
        "diet_type": DIET_TO_DIET_TYPE[diet],
    }

    max_budget = BUDGET_MAX[budget]
    candidates = [c for c in GOAL_PREFERRED_SUPPLEMENTS[goal] if c in SUPPLEMENTS]

    recommendations = []
    hard_excluded_count = 0
    for supplement in candidates:
        pred, proba = predict_effectiveness(user, supplement)
        effectiveness_level = EFFECTIVENESS_LABEL[pred]

        drug_warnings = check_drug_interactions(supplement, medications)
        allergen_warnings = check_allergens(supplement, allergies)
        diet_warnings = check_diet_compatibility(supplement, diet)
        condition_warnings = check_condition_safety(supplement, conditions)
        matched_products = find_matching_products(supplement, max_budget)

        score, tier, hard_excluded = score_and_tier(
            effectiveness_level,
            drug_warnings,
            allergen_warnings,
            diet_warnings,
            condition_warnings,
            len(matched_products) > 0,
        )

        if hard_excluded:
            hard_excluded_count += 1

```

```

        continue

    recommendations.append({
        "supplement": supplement.replace("_", " "),
        "icon": SUPPLEMENT_ICON.get(supplement, "■"),
        "effectiveness_level": effectiveness_level,
        "effectiveness_probability": proba,
        "drug_warnings": drug_warnings,
        "allergy_warnings": allergen_warnings,
        "diet_warnings": diet_warnings,
        "condition_warnings": condition_warnings,
        "matched_products": matched_products,
        "recommendation_score": score,
        "tier": tier,
    })

recommendations.sort(key=lambda r: r["recommendation_score"], reverse=True)

input_warnings = []
if conditions:
    input_warnings.append(
        "Common condition-supplement cautions were screened. Ask a clinician about conditions "
        "not covered by this safety catalogue."
    )
if not medications:
    input_warnings.append("No medication list was supplied, so medicine-interaction screening was limited.")
if not allergies:
    input_warnings.append("No allergy list was supplied, so allergen screening was limited.")

return {
    "success": True,
    "goal": goal,
    "diet": diet,
    "budget": budget,
    "training_type": user["training_type"],
    "diet_type": user["diet_type"],
    "recommendations": recommendations,
    "excluded_count": hard_excluded_count,
    "input_warnings": input_warnings,
    "disclaimer": "Research decision-support output only; not medical advice.",
}

if __name__ == "__main__":
    try:
        input_arg = sys.argv[1]

        if input_arg.endswith(".json"):
            with open(input_arg, "r", encoding="utf-8") as file:
                request_data = json.load(file)
        else:
            request_data = json.loads(input_arg)

        result = advise_supplements(request_data)
        print(json.dumps(result))

    except Exception as e:
        error_result = {
            "success": False,
            "error": str(e),
        }
        print(json.dumps(error_result))

```

ress/train_supplement_model.py

How this code works

- Reads fitness_supplements_dataset.csv and maps satisfaction to Low, Medium, or High.
- Fits categorical encoders and a balanced RandomForestClassifier, then saves both artifacts.

```

"""
Train the FITSHIELD Smart Supplement Advisor Random Forest model.
Mirrors the structure of train_model.py (Food/Drug Checker).
Outputs: ress/models/supplement_effectiveness_model.pkl, ress/models/supplement_encoders.pkl
"""
import pandas as pd
import joblib
from pathlib import Path

```

```

from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder

BASE_DIR = Path(__file__).resolve().parent
MODEL_DIR = BASE_DIR / "models"
MODEL_DIR.mkdir(exist_ok=True)

TRAIN_PATH = BASE_DIR / "datasets" / "fitness_supplements_dataset.csv"

CATEGORICAL_COLS = [
    "Gender",
    "Age",
    "Height(cm)",
    "Fitness_Level",
    "Weekly_Training",
    "Training_Type",
    "Supplement",
    "Supplement_Type",
    "Diet_Type",
]
NUMERIC_COLS = ["Weight(kg)", "Body_Fat(%)"]
FEATURES = CATEGORICAL_COLS + NUMERIC_COLS
TARGET = "Effectiveness_Level"

def bucket_effectiveness(satisfaction):
    if satisfaction < 7:
        return 0 # Low
    if satisfaction < 9:
        return 1 # Medium
    return 2 # High

print("Loading training data...")
df = pd.read_csv(TRAIN_PATH)
df[TARGET] = df["Satisfaction(1-10)"].apply(bucket_effectiveness)
print(f" {len(df)} rows, label distribution: {df[TARGET].value_counts().to_dict()}")

print("Fitting encoders...")
encoders = {}
for col in CATEGORICAL_COLS:
    le = LabelEncoder()
    df[col] = df[col].astype(str)
    le.fit(df[col])
    df[col] = le.transform(df[col])
    encoders[col] = le

X = df[FEATURES].fillna(0)
y = df[TARGET]

print("Training Random Forest (class_weight=balanced)...")
model = RandomForestClassifier(
    n_estimators=100,
    class_weight="balanced",
    random_state=42,
    n_jobs=-1,
)
model.fit(X, y)

acc = model.score(X, y)
print(f" Training accuracy: {acc:.4f}")

model_path = MODEL_DIR / "supplement_effectiveness_model.pkl"
encoder_path = MODEL_DIR / "supplement_encoders.pkl"
joblib.dump(model, model_path)
joblib.dump(encoders, encoder_path)
print(f"Saved model -> {model_path}")
print(f"Saved encoders -> {encoder_path}")
print("Done.")

```

ress/services/api-gateway/server.js

How this code works

- Routes the /api/supplement path to the supplement service without stripping the URL prefix.

```

const express = require("express");
const cors = require("cors");
const dotenv = require("dotenv");
const { createProxyMiddleware } = require("http-proxy-middleware");

```

```

dotenv.config();

const app = express();

app.use(cors());

app.get("/", (req, res) => {
  res.json({ message: "FITSHIELD API Gateway is running" });
});

// Use pathFilter (v3 syntax) at root level so Express never strips the prefix.
// The full path (e.g. /api/auth/login) reaches each service unchanged.

app.use(
  createProxyMiddleware({
    target: process.env.AUTH_SERVICE_URL,
    changeOrigin: true,
    pathFilter: "/api/auth",
  })
);

app.use(
  createProxyMiddleware({
    target: process.env.DIET_SERVICE_URL,
    changeOrigin: true,
    pathFilter: "/api/diet",
  })
);

app.use(
  createProxyMiddleware({
    target: process.env.EXERCISE_SERVICE_URL,
    changeOrigin: true,
    pathFilter: "/api/exercise",
  })
);

app.use(
  createProxyMiddleware({
    target: process.env.SUPPLEMENT_SERVICE_URL || process.env.EXERCISE_SERVICE_URL,
    changeOrigin: true,
    pathFilter: "/api/supplement",
  })
);

app.use(
  createProxyMiddleware({
    target: process.env.HABIT_SERVICE_URL,
    changeOrigin: true,
    pathFilter: "/api/habit",
  })
);

const PORT = process.env.PORT || 5001;

app.listen(PORT, "0.0.0.0", () => {
  console.log(`API Gateway running on http://0.0.0.0:${PORT}`);
});

```

6. Viva questions and model answers

Q: Why use both ML and deterministic rules?

A: The model estimates learned effectiveness from profile and supplement features. Rules are transparent and handle known medication, allergen, diet, condition, and budget constraints. Combining them is safer and easier to explain than relying on either alone.

Q: Why is the advice endpoint optional-authenticated?

A: A visitor can receive general advice without signing in, but a valid Bearer token lets the controller link the generated advice to that user's history.

Q: What does the model predict?

A: It predicts an effectiveness class: Low, Medium, or High. This is separate from the final safety tier, which is calculated after warnings and product availability are considered.

Q: Why save the encoders with the model?

A: The model expects the exact numerical mappings used during training. Reusing the saved encoders keeps inference consistent after a restart.

Q: What happens when a medication interaction is found?

A: The interaction is returned as structured warning data. A high-severity interaction can hard-exclude the supplement and force a Not Recommended tier.

Q: How are allergies handled?

A: Free-text allergy input is normalized and expanded using alias groups, then matched against supplement allergen and category fields. The response records the allergen, severity, reaction, and recommendation.

Q: How do vegan and vegetarian restrictions work?

A: The diet rule table marks animal-derived products such as fish oil or dairy-derived whey as incompatible and supplies a safer substitute recommendation.

Q: How is the budget enforced?

A: The selected budget maps to a maximum LKR amount. Products above that maximum are removed; the implementation does not silently fall back to over-budget products.

Q: Why are products de-duplicated?

A: The catalog can contain repeated brand/product rows. The service keeps the highest-rated row for each normalized brand and product, then returns at most three matches.

Q: How is the final score calculated?

A: The score starts at 30, adds effectiveness points and an in-budget product bonus, subtracts warning penalties, clamps to 0-100, and maps to Recommended, Caution, or Not Recommended.

Q: What happens if the model file is missing?

A: The Python process returns a failure JSON because real inference cannot run. Regenerate the ignored artifacts with `train_supplement_model.py` before using the endpoint.

Q: How are Python errors handled by Node?

A: The controller listens for process errors and close status, parses only valid JSON, and returns an explicit HTTP error instead of pretending advice succeeded.

Q: What is the target variable?

A: The training script derives Effectiveness_Level from Satisfaction(1-10): below 7 is Low, 7-8 is Medium, and 9-10 is High.

Q: Why use class_weight='balanced'?

A: If one effectiveness class has more rows than another, balancing gives minority classes more influence and reduces the chance that accuracy hides poor minority-class behavior.

Q: What is a Random Forest?

A: It is an ensemble of decision trees trained with variation in samples and features. Their combined votes usually generalize better than one tree.

Q: What is model inference?

A: Inference is applying the already-trained model and its encoders to a new normalized profile and supplement row. It is different from fitting the model.

Q: How is supplement advice persisted?

A: When req.user exists, the controller creates a SupplementAdviceLog with the user's context, recommendations, and excluded count. History endpoints query records by userId.

Q: Why are saved supplements separate from advice history?

A: Advice history is an immutable record of generated outputs, while the saved list is a user-managed collection that can be replaced independently.

Q: How would you improve this ML system?

A: Add more representative labeled outcomes, evaluate on held-out data, monitor class-specific recall, version models and encoders together, and validate safety rules against clinician-reviewed sources.

7. Demonstration checklist

- Use a sample profile such as Male, age 28, height 175 cm, weight 72 kg, Intermediate, 3-5 weekly sessions, Muscle Gain, Non-Vegetarian, and No Limit.
- Show the request body fields: goal, age, weight, height, gender, diet, budget, medications, allergies, and conditions.
- Point out the final goal, training_type, diet_type, ranked recommendations, effectiveness level, probability, score, tier, matched products, and input warnings.
- Demonstrate a medication or condition warning, for example a stimulant supplement with high blood pressure, and explain why it can be excluded.
- Demonstrate a vegan or vegetarian restriction and show the replacement recommendation in the diet warning.
- Show that a tight budget removes products over the selected LKR ceiling rather than displaying an over-budget fallback.
- Show authenticated history, saved supplements, and progress-note endpoints after signing in.
- Open train_supplement_model.py and point out the satisfaction-to-class mapping, encoders, balanced RandomForestClassifier, and saved model files.

Example response shape

```
{
  "success": true,
  "goal": "Muscle Gain",
  "diet": "Non-Vegetarian",
  "budget": "No Limit",
  "training_type": "Strength_Training",
  "recommendations": [{
    "supplement": "Whey Protein",
    "effectiveness_level": "High",
    "drug_warnings": [],
    "allergy_warnings": [],
    "matched_products": [],
    "recommendation_score": 70,
    "tier": "Recommended"
  }],
  "excluded_count": 0,
  "input_warnings": [],
  "disclaimer": "Research decision-support output only; not medical advice."
}
```

8. Run commands

Install the shared Node dependencies and Python ML dependencies through the project environment. Generate the ignored supplement model artifacts from the checked-in training data, then start the complete system. The browser is served on port 5000 and the supplement service listens on port 5006 behind the API gateway.

```
npm install
python3 -m pip install -r pyproject.toml
cd ress
python3 train_supplement_model.py
cd ..
bash start.sh
```

Required runtime details

- The supplement service receives its port from PORT and defaults to 5006.
- The gateway forwards /api/supplement to SUPPLEMENT_SERVICE_URL.
- JWT_SECRET is supplied by the startup environment so authenticated history can be linked to a user.
- MONGO_URI is optional for public advice, but it is required for durable accounts, advice history, saved supplements, and progress notes.
- The trained files are supplement_effectiveness_model.pkl and supplement_encoders.pkl under ress/models.

The endpoint is a decision-support feature and does not replace professional medical advice.