Python Bootcamp - Session 4

Python Integration with Other Technologies

Hrishikesh Terdalkar

November 30, 2025

Session Overview

Session 4 builds on the shared datasets from the previous session and demonstrates how to integrate them with databases, external data sources, parallel compute, and a lightweight dashboard. The goal is to provide a complete reference workflow that you can adapt to your own research problems.

What You Will Build

- A SQLite-backed store of the Session 3 datasets using SQLAlchemy models.
- A small API collector that fetches (or mocks) external context and analyses it.
- A parallel Monte Carlo benchmark with a side-by-side sequential comparison.
- A minimal Flask dashboard that exposes analysis results as HTML and JSON.
- An integration script that ties all pieces together and writes a report.

Session Plan

- Database ingest and quick statistics.
- API integration (mock weather/material properties) and CSV export.
- Parallel processing demo and speedup discussion.
- Dashboard walkthrough (CLI mode first, optional Flask run).
- Integration demo and report generation.

Skills You'll Practice

- Persisting experiment data with SQLAlchemy models (SQLite).
- Designing small API clients, handling JSON, and saving tidy CSVs.
- Running CPU-bound work in parallel and interpreting speedups.
- Exposing analysis via a minimal Flask dashboard and JSON routes.
- Integrating outputs across tools and generating a final report.

Learning Outcomes

By the end of this session, you should be able to:

- Create and query a SQLite database using SQLAlchemy ORM classes.
- Collect or mock external API data and analyze simple trends.
- Compare sequential vs parallel runs and reason about efficiency.
- Serve analyses over HTTP (Flask) and export artefacts on demand.
- Produce an end-to-end integration report from combined components.

Theory Essentials

- **ORM basics**: SQLAlchemy maps Python classes to tables so you can query without handwritten SQL.
- **API fundamentals**: Keep authentication and headers centralised; persist raw responses before transforming.
- **Parallelism**: Use processes for CPU-bound workloads; threads help when the bottleneck is I/O.
- **Amdahl's law**: Speedup is limited by the serial fraction; profile before parallelising.
- **Web architecture**: Separate routes (HTTP) from analysis functions so you can test logic without a server.

Prerequisites and Setup

Use the same environment from Session 3 or create a fresh one. The examples assume you have already generated the shared dataset with `make data`.

```
# Install dependencies into your active environment
pip install -r requirements.txt

# Prepare shared data produced in Session 3
make data
```

How to Run the Examples

1. Persist CSV data to SQLite:

```
python session4/01_database.py
# Creates research_data.db and writes statistics + CSV export

```

2. Collect and analyze API data (mocked for offline use):

```
python session4/02_api_integration.py
# Produces weather_analysis_data.csv for later use

```

3. Compare sequential vs parallel workloads:

```
python session4/03_parallel_processing.py
# Observe speedup and verify identical results across modes

```

4. Explore the dashboard (demo or server):

```
python session4/04_research_dashboard.py
# or run a local server
flask -{}-app session4/04_research_dashboard.py run -{}-port 5000

```

5. Tie everything together and generate a report:

```
python session4/05_integration_demo.py
# Writes integrated_system_report.json

```

Lab Guide

01_database.py

- Defines SQLAlchemy models and a helper class to ingest DataFrames.
- Adds data in batch and computes basic per-parameter statistics.
- Exports an experiment to CSV for downstream tools.

02_api_integration.py

- Demonstrates a simple API client pattern with a session and headers.
- Uses the free Open-Meteo weather API when the network is available, and falls back to deterministic mock data otherwise.
- Produces a tidy CSV for reuse in plotting or dashboards.

03_parallel_processing.py

- Benchmarks sequential vs parallel Monte Carlo simulations.
- Shows chunked batch processing with and without processes.
- Emphasises correctness: compare results across modes before celebrating speed.

04_research_dashboard.py

- Minimal Flask app exposing analysis as HTML/JSON.
- Generates plots in-memory and returns base64-encoded PNGs.
- Includes endpoints to analyze, plot, export, and list experiments.

05_integration_demo.py

- Generates a small dataset, writes it to CSV and SQLite, and combines it with mock API data.
- Runs a few parallel jobs, creates plots, and writes a concise JSON integration report.

Best Practices

0.1 Error Handling

- Database: check connections, wrap commits in `try/except`, and roll back on failure.
- APIs: set timeouts, catch `requests` exceptions, and log raw payloads before transforming.
- Parallel jobs: handle exceptions inside worker functions and return error details to the parent.
- Flask: validate inputs from `request.json` and return proper status codes.

0.2 Documentation

- Document ORM models (field purpose/units) and any expected CSV schema.
- Record API endpoints, auth method, and rate limits near the client class.
- Include CLI examples (copy/paste) for each module; keep `–help` informative.

Common Pitfalls

- Activate your virtual environment before running the Flask CLI; otherwise `flask` may not find dependencies.
- If `engineering_test_data.csv` is missing, run `make data` to regenerate the shared dataset.
- Firewall prompts can block the Flask server from binding to a port; allow local connections if prompted.

More Examples

Persist correlations after database ingest:

```
df = db.get_experiment_data("thermal_study_001")
num = df.pivot_table(index="timestamp", columns="parameter", values="value")
num.corr().to_csv("correlations.csv")
```

Simple parallel parameter sweep skeleton:

```
from concurrent.futures import ProcessPoolExecutor
def simulate(param):
    # compute something CPU-bound
    return param, param**2
with ProcessPoolExecutor() as ex:
    results = list(ex.map(simulate, range(8)))
```

Practice Exercises

1. Add a new SQLAlchemy model that stores computed rolling means, then write a short script to populate it.
2. Replace the mock weather generator with a free API of your choice (for example, Open-Meteo), and save raw JSON before flattening to CSV.
3. Extend the dashboard with one extra route that returns a correlation heatmap as a PNG.

Further Reading

- Official Python documentation: `https://docs.python.org/3/`
- pandas documentation (data analysis): `https://pandas.pydata.org/docs/`
- SQLAlchemy documentation (databases): `https://docs.sqlalchemy.org/`
- Flask documentation (web apps): `https://flask.palletsprojects.com/`
- Open-Meteo API docs (example of a free, research-friendly API): `https://open-meteo.com/en/docs`

Appendix: Full Code Listings

session4/01_database.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Research Database Integration (advanced / optional)

This script uses SQLAlchemy (an Object-Relational Mapper)
to talk to a SQLite database. It is here to show what a
more 'real world' database layer can look like.

If you are new to Python, you do not need to understand
every line - you can simply run the script once to see
that experiments and data points are stored in a database.

@author: Hrishikesh Terdalkar
"""

###############################################################################

import json
from datetime import datetime
from pathlib import Path

import numpy as np
import pandas as pd
from sqlalchemy import (
    create_engine,
    Column,
    Integer,
    String,
    Float,
    DateTime,
    Text,
)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

###############################################################################

RNG_SEED = 42
np.random.seed(RNG_SEED)

Base = declarative_base()


class ResearchExperiment(Base):
    """ORM class for research experiments"""

    __tablename__ = "research_experiments"

    id = Column(Integer, primary_key=True)
    experiment_id = Column(String(100), unique=True, nullable=False)
    title = Column(String(200), nullable=False)
    description = Column(Text)
    researcher = Column(String(100))
    start_date = Column(DateTime)
    end_date = Column(DateTime)
    created_at = Column(DateTime, default=datetime.now)

    def __repr__(self):
        return f"<Experiment(id={self.experiment_id}, title='{self.title}')>"


class ExperimentalData(Base):
    """ORM class for experimental data points"""

    __tablename__ = "experimental_data"

    id = Column(Integer, primary_key=True)
    experiment_id = Column(String(100), nullable=False)
```

```
    timestamp = Column(DateTime, nullable=False)
    parameter_name = Column(String(100), nullable=False)
    parameter_value = Column(Float, nullable=False)
    uncertainty = Column(Float)
    units = Column(String(50))
    notes = Column(Text)

    def __repr__(self):
        return f"<DataPoint(exp={self.experiment_id}, param={self.parameter_name}, value={self
    .parameter_value})>"


class ResearchDatabase:
    """Database management class for research data"""

    def __init__(self, db_url="sqlite:///research_data.db"):
        self.engine = create_engine(db_url)
        Base.metadata.create_all(self.engine)
        Session = sessionmaker(bind=self.engine)
        self.session = Session()

    def create_experiment(
        self,
        experiment_id,
        title,
        researcher,
        description="",
        start_date=None,
        end_date=None,
    ):
        """Create a new experiment record

        If an experiment with the same ``experiment_id`` already exists,
        return the existing record instead of raising an error. This makes
        the demonstration script safe to run multiple times.
        """
        # Check for existing experiment first (idempotent behaviour for the demo)
        existing = (
            self.session.query(ResearchExperiment)
            .filter_by(experiment_id=experiment_id)
            .first()
        )
        if existing is not None:
            return existing

        experiment = ResearchExperiment(
            experiment_id=experiment_id,
            title=title,
            researcher=researcher,
            description=description,
            start_date=start_date or datetime.now(),
            end_date=end_date,
        )
        self.session.add(experiment)
        self.session.commit()
        return experiment

    def add_data_point(
        self,
        experiment_id,
        parameter_name,
        parameter_value,
        timestamp=None,
        uncertainty=None,
        units="",
        notes="",
    ):
        """Add a single data point"""
        data_point = ExperimentalData(
            experiment_id=experiment_id,
            parameter_name=parameter_name,
            parameter_value=parameter_value,
            timestamp=timestamp or datetime.now(),
            uncertainty=uncertainty,
            units=units,
```

```
            notes=notes,
        )
        self.session.add(data_point)
        self.session.commit()
        return data_point

    def add_data_batch(self, experiment_id, data_frame, time_column="time"):
        """Add multiple data points from a DataFrame"""
        data_points = []

        for idx, row in data_frame.iterrows():
            if time_column in data_frame.columns:
                timestamp = pd.to_datetime(row[time_column])
                if isinstance(timestamp, pd.Timestamp):
                    timestamp = timestamp.to_pydatetime()
            else:
                timestamp = datetime.now()

            for col in data_frame.columns:
                if col != time_column and pd.api.types.is_numeric_dtype(
                    data_frame[col]
                ):
                    data_point = ExperimentalData(
                        experiment_id=experiment_id,
                        parameter_name=col,
                        parameter_value=float(row[col]),
                        timestamp=timestamp,
                    )
                    data_points.append(data_point)

        self.session.bulk_save_objects(data_points)
        self.session.commit()
        return len(data_points)

    def get_experiment_data(self, experiment_id, parameter_name=None):
        """Retrieve data for a specific experiment"""
        query = self.session.query(ExperimentalData).filter_by(
            experiment_id=experiment_id
        )

        if parameter_name:
            query = query.filter_by(parameter_name=parameter_name)

        results = query.all()

        # Convert to DataFrame for analysis
        if results:
            data = []
            for result in results:
                data.append(
                    {
                        "timestamp": result.timestamp,
                        "parameter": result.parameter_name,
                        "value": result.parameter_value,
                        "uncertainty": result.uncertainty,
                        "units": result.units,
                    }
                )
            return pd.DataFrame(data)
        else:
            return pd.DataFrame()

    def get_experiment_statistics(self, experiment_id):
        """Calculate statistics for an experiment"""
        data = self.get_experiment_data(experiment_id)

        if data.empty:
            return None

        statistics = {}
        parameters = data["parameter"].unique()

        for param in parameters:
            param_data = data[data["parameter"] == param]["value"]
            statistics[param] = {
```

```
                "count": len(param_data),
                "mean": float(param_data.mean()),
                "std": float(param_data.std()),
                "min": float(param_data.min()),
                "max": float(param_data.max()),
            }

        return statistics

    def export_experiment_to_csv(self, experiment_id, output_file):
        """Export experiment data to CSV"""
        data = self.get_experiment_data(experiment_id)

        if not data.empty:
            data.to_csv(output_file, index=False)
            return True
        return False

    def list_experiments(self):
        """List all experiments in the database"""
        return self.session.query(ResearchExperiment).all()


def demonstrate_database_operations():
    """Demonstrate database operations with sample data"""
    print("=== RESEARCH DATABASE DEMONSTRATION ===")

    # Initialize database
    db = ResearchDatabase()

    # Create sample experiment
    experiment = db.create_experiment(
        experiment_id="thermal_study_001",
        title="Thermal Conductivity Measurement",
        researcher="PhD Student",
        description="Measuring thermal conductivity of composite materials",
    )
    print(f"Created experiment: {experiment}")

    csv_path = Path("engineering_test_data.csv")
    if csv_path.exists():
        thermal_df = pd.read_csv(csv_path)
        if "time" not in thermal_df.columns:
            anchor = pd.Timestamp("2024-01-01 09:00:00")
            if "Time_min" in thermal_df.columns:
                thermal_df["time"] = anchor + pd.to_timedelta(
                    thermal_df["Time_min"], unit="m"
                )
            else:
                thermal_df["time"] = pd.date_range(
                    start=anchor, periods=len(thermal_df), freq="min"
                )
        print(f"Loaded dataset from {csv_path}")
    else:
        time_points = pd.date_range(start="2024-01-01", periods=100, freq="H")
        temperatures = (
            25
            + 10 * np.sin(2 * np.pi * np.arange(100) / 24)
            + np.random.normal(0, 1, 100)
        )
        heat_flux = (
            100
            + 20 * np.cos(2 * np.pi * np.arange(100) / 12)
            + np.random.normal(0, 5, 100)
        )

        thermal_df = pd.DataFrame(
            {
                "time": time_points,
                "temperature": temperatures,
                "heat_flux": heat_flux,
            }
        )
        print("Generated synthetic thermal dataset for demonstration")
```

```
    # Add data batch
    points_added = db.add_data_batch("thermal_study_001", thermal_df, "time")
    print(f"Added {points_added} data points")

    # Retrieve and analyze data
    data = db.get_experiment_data("thermal_study_001")
    print(f"Retrieved data shape: {data.shape}")

    statistics = db.get_experiment_statistics("thermal_study_001")
    print("\nExperiment Statistics:")
    for param, stats in statistics.items():
        print(f"{param}: mean={stats['mean']:.2f} +/- {stats['std']:.2f}")

    # List all experiments
    experiments = db.list_experiments()
    print(f"\nTotal experiments in database: {len(experiments)}")

    # Export to CSV
    db.export_experiment_to_csv("thermal_study_001", "thermal_data_export.csv")
    print("Data exported to thermal_data_export.csv")


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

Listing 1: SQLAlchemy database integration

session4/02_api_integration.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
API Integration for Research Data (intermediate)

This file shows a small, structured way to:
- call a free weather API (Open-Meteo) when network access is available
- fall back to local mock data if the request fails
- turn JSON-like responses into pandas DataFrames
- do a bit of trend analysis.

It also includes a small material-properties client that uses only
local mock data, plus a template for APIs that expect bearer tokens.

Treat the helper classes as ready-made tools - the main focus is the
high-level flow in `demonstrate_api_integration()`.

@author: Hrishikesh Terdalkar
"""

###############################################################################

import time
import json
from datetime import datetime, timedelta

import requests
import pandas as pd
import numpy as np

###############################################################################

RNG_SEED = 42
np.random.seed(RNG_SEED)


class ResearchDataAPI:
    """Base class for research data API integration"""

    def __init__(self, base_url=None, api_key=None):
        self.base_url = base_url
```

```
        self.api_key = api_key
        self.session = requests.Session()

        # Common headers for API requests
        self.headers = {
            "User-Agent": "Research Data Collector/1.0",
            "Accept": "application/json",
        }

        if api_key:
            self.headers["Authorization"] = f"Bearer {api_key}"

    def make_request(self, endpoint, params=None, method="GET"):
        """Make API request with error handling"""
        url = f"{self.base_url}/{endpoint}" if self.base_url else endpoint

        try:
            if method == "GET":
                response = self.session.get(
                    url, params=params, headers=self.headers
                )
            elif method == "POST":
                response = self.session.post(
                    url, json=params, headers=self.headers
                )
            else:
                raise ValueError(f"Unsupported method: {method}")

            response.raise_for_status()  # Raise exception for bad status codes
            return response.json()

        except requests.exceptions.RequestException as e:
            print(f"API request failed: {e}")
            return None

    def rate_limit_delay(self, delay_seconds=1):
        """Simple rate limiting"""
        time.sleep(delay_seconds)


class WeatherDataCollector(ResearchDataAPI):
    """Collect weather data for environmental studies"""

    def __init__(self, api_key=None):
        # Use the free Open-Meteo API for live data when possible.
        # Documentation: https://open-meteo.com/en/docs
        super().__init__("https://api.open-meteo.com/v1", api_key)

    @staticmethod
    def _city_to_coordinates(city, country_code=None):
        """Map a few example cities to coordinates required by Open-Meteo"""
        key = f"{city},{country_code}".lower() if country_code else city.lower()
        mapping = {
            "london,uk": (51.5074, -0.1278),
            "london": (51.5074, -0.1278),
            "mumbai,in": (19.0760, 72.8777),
            "mumbai": (19.0760, 72.8777),
            "hyderabad,in": (17.3850, 78.4867),
            "hyderabad": (17.3850, 78.4867),
        }
        return mapping.get(key, (51.5074, -0.1278))  # default: London

    def get_current_weather(self, city, country_code=None):
        """Get current weather data using Open-Meteo, with mock fallback"""
        lat, lon = self._city_to_coordinates(city, country_code)

        params = {
            "latitude": lat,
            "longitude": lon,
            "current_weather": "true",
        }

        raw = self.make_request("forecast", params=params, method="GET")

        if raw and "current_weather" in raw:
```

```
            cw = raw["current_weather"]
            processed = {
                "city": city,
                "temperature": cw.get("temperature"),
                "pressure": None,  # Open-Meteo current_weather does not include pressure
                "humidity": None,
                "wind_speed": cw.get("windspeed"),
                "conditions": "N/A",
                "timestamp": datetime.fromisoformat(
                    cw.get("time")
                ).isoformat(),
            }
            return processed

        # Fallback: local mock if API is unreachable or response incomplete
        mock_data = {
            "weather": [{"main": "Clear", "description": "clear sky"}],
            "main": {
                "temp": 15.5 + np.random.normal(0, 3),
                "pressure": 1013 + np.random.normal(0, 5),
                "humidity": 65 + np.random.normal(0, 10),
                "temp_min": 13.0,
                "temp_max": 18.0,
            },
            "wind": {"speed": 3.1 + np.random.normal(0, 1), "deg": 240},
            "name": city,
            "dt": int(datetime.now().timestamp()),
        }

        return self._process_weather_data(mock_data)

    def get_historical_weather(self, city, days=7):
        """Generate historical weather data (mock)"""
        historical_data = []
        end_date = datetime.now()

        for i in range(days):
            current_date = end_date - timedelta(days=i)

            # Generate realistic seasonal data
            day_of_year = current_date.timetuple().tm_yday
            base_temp = 10 + 10 * np.sin(
                2 * np.pi * (day_of_year - 80) / 365
            )  # Seasonal variation

            daily_data = {
                "date": current_date.strftime("%Y-%m-%d"),
                "temperature": base_temp + np.random.normal(0, 2),
                "pressure": 1013 + np.random.normal(0, 3),
                "humidity": 60 + np.random.normal(0, 15),
                "wind_speed": 3 + abs(np.random.normal(0, 1.5)),
                "conditions": np.random.choice(
                    ["Clear", "Cloudy", "Rain", "Snow"]
                ),
            }
            historical_data.append(daily_data)

        return historical_data

    def _process_weather_data(self, raw_data):
        """Process raw API response into structured format"""
        return {
            "city": raw_data.get("name", "Unknown"),
            "temperature": raw_data["main"]["temp"],
            "pressure": raw_data["main"]["pressure"],
            "humidity": raw_data["main"]["humidity"],
            "wind_speed": raw_data["wind"]["speed"],
            "conditions": raw_data["weather"][0]["main"],
            "timestamp": datetime.fromtimestamp(raw_data["dt"]).isoformat(),
        }

    def analyze_weather_trends(self, historical_data):
        """Analyze weather trends for research"""
        df = pd.DataFrame(historical_data)
```

```
        df["date"] = pd.to_datetime(df["date"])

        analysis = {
            "analysis_period": {
                "start": df["date"].min().strftime("%Y-%m-%d"),
                "end": df["date"].max().strftime("%Y-%m-%d"),
                "days": len(df),
            },
            "temperature_analysis": {
                "mean": df["temperature"].mean(),
                "trend": (
                    "increasing"
                    if df["temperature"].iloc[-1] > df["temperature"].iloc[0]
                    else "decreasing"
                ),
                "daily_variation": df["temperature"].std(),
            },
            "pressure_analysis": {
                "mean": df["pressure"].mean(),
                "correlation_with_temp": df["temperature"].corr(
                    df["pressure"]
                ),
            },
            "condition_frequency": df["conditions"].value_counts().to_dict(),
        }

        return analysis, df


class MaterialPropertiesAPI(ResearchDataAPI):
    """Mock API for material properties data"""

    def __init__(self):
        # Uses only local mock data; URL and key are placeholders.
        super().__init__("https://example.com/materials-api", "demo_key")

        # Mock material database
        self.materials_db = {
            "aluminum": {
                "density": 2.70,  # g/cm^3
                "youngs_modulus": 69,  # GPa
                "thermal_conductivity": 237,  # W/m*K
                "specific_heat": 0.897,  # J/g*K
            },
            "steel": {
                "density": 7.85,
                "youngs_modulus": 200,
                "thermal_conductivity": 50,
                "specific_heat": 0.466,
            },
            "copper": {
                "density": 8.96,
                "youngs_modulus": 110,
                "thermal_conductivity": 401,
                "specific_heat": 0.385,
            },
        }

    def get_material_properties(self, material_name):
        """Get properties for a specific material"""
        material_name = material_name.lower()

        if material_name in self.materials_db:
            return self.materials_db[material_name]
        else:
            return {"error": f"Material '{material_name}' not found"}

    def compare_materials(self, material_list, property_name):
        """Compare specific property across materials"""
        comparison = {}

        for material in material_list:
            props = self.get_material_properties(material)
            if property_name in props:
                comparison[material] = props[property_name]
```

```
        return comparison


def demonstrate_api_integration():
    """Demonstrate API integration with research data"""
    print("=== API INTEGRATION DEMONSTRATION ===")

    # Weather data collection
    weather_collector = WeatherDataCollector()

    print("1. Current Weather Data:")
    current_weather = weather_collector.get_current_weather("London", "UK")
    for key, value in current_weather.items():
        print(f"   {key}: {value}")

    print("\n2. Historical Weather Analysis:")
    historical_data = weather_collector.get_historical_weather("London", 30)
    analysis, weather_df = weather_collector.analyze_weather_trends(
        historical_data
    )

    print(
        f"   Period: {analysis['analysis_period']['start']} to {analysis['analysis_period']['
    end']}"
    )
    print(
        f"   Mean temperature: {analysis['temperature_analysis']['mean']:.1f} deg C"
    )
    print(f"   Trend: {analysis['temperature_analysis']['trend']}")

    # Material properties API
    materials_api = MaterialPropertiesAPI()

    print("\n3. Material Properties:")
    materials = ["aluminum", "steel", "copper"]
    for material in materials:
        props = materials_api.get_material_properties(material)
        print(f"   {material.capitalize()}:")
        print(f"     Density: {props['density']} g/cm^3")
        print(f"     Young's Modulus: {props['youngs_modulus']} GPa")

    print("\n4. Material Comparison (Thermal Conductivity):")
    comparison = materials_api.compare_materials(
        materials, "thermal_conductivity"
    )
    for material, conductivity in comparison.items():
        print(f"   {material}: {conductivity} W/m*K")

    # Save data for further analysis
    weather_df.to_csv("weather_analysis_data.csv", index=False)
    print("\nWeather data saved to weather_analysis_data.csv")


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

Listing 2: API integration (mock weather and materials)

session4/03_parallel_processing.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Parallel Processing for Research (intermediate)

Goal: show how running the same calculation many times
can be sped up by using multiple CPU cores.

You do not need to understand every detail of the Monte
```

```
Carlo examples - focus on the difference between the
sequential and parallel timings printed by the script.

@author: Hrishikesh Terdalkar
"""

##############################################################################

import time
import multiprocessing as mp
from concurrent.futures import ProcessPoolExecutor

import numpy as np
import pandas as pd

##############################################################################

RNG_SEED = 42
np.random.seed(RNG_SEED)


class ResearchParallelProcessor:
    """Parallel processing utilities for research applications"""

    @staticmethod
    def get_system_info():
        """Get information about available processing resources"""
        cpu_total = mp.cpu_count()
        print("=== SYSTEM INFORMATION ===")
        print(f"CPU Cores: {cpu_total}")

        return cpu_total


class MonteCarloSimulator:
    """Parallel Monte Carlo simulation for engineering applications"""

    def __init__(self):
        self.results = []

    @staticmethod
    def monte_carlo_trial(
        trial_id, num_samples=1000, simulation_type="structural"
    ):
        """Single Monte Carlo trial - simulating different engineering scenarios"""

        if simulation_type == "structural":
            # Structural reliability simulation
            load = np.random.normal(100, 15, num_samples)  # kN - random load
            strength = np.random.normal(
                150, 20, num_samples
            )  # kN - material strength
            safety_margin = strength - load
            failures = np.sum(safety_margin < 0)
            reliability = 1 - (failures / num_samples)

            return {
                "trial_id": trial_id,
                "reliability": reliability,
                "mean_safety_margin": np.mean(safety_margin),
                "failure_probability": failures / num_samples,
            }

        elif simulation_type == "thermal":
            # Thermal analysis simulation
            initial_temp = np.random.normal(20, 2, num_samples)
            heat_input = np.random.normal(1000, 100, num_samples)
            material_resistance = np.random.normal(0.5, 0.1, num_samples)
            final_temp = initial_temp + heat_input * material_resistance

            # Check for overheating
            overheat_count = np.sum(final_temp > 100)

            return {
                "trial_id": trial_id,
```

```
                "mean_final_temp": np.mean(final_temp),
                "overheat_probability": overheat_count / num_samples,
                "temp_std": np.std(final_temp),
            }

        else:
            raise ValueError(f"Unknown simulation type: {simulation_type}")

    def run_sequential_simulation(
        self, num_trials=100, num_samples=1000, simulation_type="structural"
    ):
        """Run simulation sequentially for comparison"""
        print(f"Running {num_trials} {simulation_type} trials sequentially...")

        start_time = time.time()
        results = []

        for i in range(num_trials):
            result = self.monte_carlo_trial(i, num_samples, simulation_type)
            results.append(result)

        sequential_time = time.time() - start_time
        return results, sequential_time

    def run_parallel_simulation(
        self,
        num_trials=100,
        num_samples=1000,
        simulation_type="structural",
        num_workers=None,
    ):
        """Run simulation in parallel"""
        if num_workers is None:
            num_workers = mp.cpu_count()

        print(
            f"Running {num_trials} {simulation_type} trials in parallel ({num_workers} workers
    )..."
        )

        start_time = time.time()

        with ProcessPoolExecutor(max_workers=num_workers) as executor:
            futures = [
                executor.submit(
                    self.monte_carlo_trial, i, num_samples, simulation_type
                )
                for i in range(num_trials)
            ]
            results = [future.result() for future in futures]

        parallel_time = time.time() - start_time
        return results, parallel_time

    def analyze_simulation_results(self, results, simulation_type):
        """Analyze and summarize simulation results"""
        if simulation_type == "structural":
            reliabilities = [r["reliability"] for r in results]
            safety_margins = [r["mean_safety_margin"] for r in results]

            summary = {
                "simulation_type": simulation_type,
                "trials": len(results),
                "mean_reliability": np.mean(reliabilities),
                "reliability_std": np.std(reliabilities),
                "mean_safety_margin": np.mean(safety_margins),
                "reliability_95_ci": [
                    np.percentile(reliabilities, 2.5),
                    np.percentile(reliabilities, 97.5),
                ],
            }

        elif simulation_type == "thermal":
            final_temps = [r["mean_final_temp"] for r in results]
            overheat_probs = [r["overheat_probability"] for r in results]
```

```
            summary = {
                "simulation_type": simulation_type,
                "trials": len(results),
                "mean_final_temp": np.mean(final_temps),
                "overheat_probability": np.mean(overheat_probs),
                "temp_variation": np.std(final_temps),
            }

        return summary


class DataBatchProcessor:
    """Parallel processing for batch data operations"""

    @staticmethod
    def process_data_chunk(chunk_data, operation="statistics"):
        """Process a chunk of data with specified operation"""

        if operation == "statistics":
            return {
                "chunk_size": len(chunk_data),
                "mean": np.mean(chunk_data),
                "std": np.std(chunk_data),
                "min": np.min(chunk_data),
                "max": np.max(chunk_data),
            }

        elif operation == "fft":
            # Simulate FFT analysis
            fft_result = np.fft.fft(chunk_data)
            magnitude = np.abs(fft_result)
            return {
                "chunk_size": len(chunk_data),
                "dominant_frequency": np.argmax(
                    magnitude[1 : len(magnitude) // 2]
                )
                + 1,
                "max_magnitude": np.max(magnitude),
            }

        else:
            raise ValueError(f"Unknown operation: {operation}")

    def process_large_dataset(
        self, data, chunk_size=1000, operation="statistics", parallel=True
    ):
        """Process large dataset in chunks, optionally in parallel"""
        chunks = [
            data[i : i + chunk_size] for i in range(0, len(data), chunk_size)
        ]
        print(f"Processing {len(data)} points in {len(chunks)} chunks...")

        start_time = time.time()

        if parallel:
            with ProcessPoolExecutor() as executor:
                futures = [
                    executor.submit(self.process_data_chunk, chunk, operation)
                    for chunk in chunks
                ]
                results = [future.result() for future in futures]
        else:
            results = [
                self.process_data_chunk(chunk, operation) for chunk in chunks
            ]

        processing_time = time.time() - start_time
        return results, processing_time


def demonstrate_parallel_processing():
    """Demonstrate parallel processing capabilities"""
    print("=== PARALLEL PROCESSING DEMONSTRATION ===")

```

```
    # System information
    cpu_count = ResearchParallelProcessor.get_system_info()

    # Monte Carlo simulation comparison
    simulator = MonteCarloSimulator()

    print("\n1. Monte Carlo Simulation - Structural Reliability")
    print("-" * 50)

    # Sequential simulation
    seq_results, seq_time = simulator.run_sequential_simulation(
        num_trials=50, num_samples=5000, simulation_type="structural"
    )

    # Parallel simulation
    par_results, par_time = simulator.run_parallel_simulation(
        num_trials=50, num_samples=5000, simulation_type="structural"
    )

    # Performance comparison
    speedup = seq_time / par_time
    efficiency = (speedup / cpu_count) * 100

    print(f"Sequential time: {seq_time:.2f} seconds")
    print(f"Parallel time: {par_time:.2f} seconds")
    print(f"Speedup: {speedup:.2f}x")
    print(f"Efficiency: {efficiency:.1f}%")

    # Analyze results
    summary = simulator.analyze_simulation_results(par_results, "structural")
    print(f"\nSimulation Results:")
    print(f"Mean reliability: {summary['mean_reliability']:.3f}")
    print(f"Reliability std: {summary['reliability_std']:.3f}")
    print(
        f"95% CI: [{summary['reliability_95_ci'][0]:.3f}, {summary['reliability_95_ci'][1]:.3f
    }]"
    )

    # Thermal simulation
    print("\n2. Monte Carlo Simulation - Thermal Analysis")
    print("-" * 50)

    thermal_results, thermal_time = simulator.run_parallel_simulation(
        num_trials=30, simulation_type="thermal"
    )

    thermal_summary = simulator.analyze_simulation_results(
        thermal_results, "thermal"
    )
    print(f"Parallel time: {thermal_time:.2f} seconds")
    print(
        f"Mean final temperature: {thermal_summary['mean_final_temp']:.1f} deg C"
    )
    print(
        f"Overheat probability: {thermal_summary['overheat_probability']:.3f}"
    )

    # Batch data processing
    print("\n3. Batch Data Processing")
    print("-" * 50)

    processor = DataBatchProcessor()

    # Generate large dataset
    large_data = np.random.normal(0, 1, 50000)  # 50,000 data points

    # Sequential processing
    seq_batch, seq_batch_time = processor.process_large_dataset(
        large_data, chunk_size=1000, operation="statistics", parallel=False
    )

    # Parallel processing
    par_batch, par_batch_time = processor.process_large_dataset(
        large_data, chunk_size=1000, operation="statistics", parallel=True
```

```
    )

    batch_speedup = seq_batch_time / par_batch_time

    print(f"Dataset size: {len(large_data)} points")
    print(f"Sequential batch time: {seq_batch_time:.2f} seconds")
    print(f"Parallel batch time: {par_batch_time:.2f} seconds")
    print(f"Batch processing speedup: {batch_speedup:.2f}x")

    # Verify results are consistent
    seq_means = [chunk["mean"] for chunk in seq_batch]
    par_means = [chunk["mean"] for chunk in par_batch]

    if np.allclose(seq_means, par_means, rtol=1e-10):
        print("[OK] Sequential and parallel results are identical")
    else:
        print(
            "[WARN] Results differ between sequential and parallel processing"
        )


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

Listing 3: Parallel processing and Monte Carlo

session4/04_research_dashboard.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Research Dashboard with Flask (advanced / optional)

This script is a larger example that combines several
ideas from the rest of Session 4 into a small web app.

If you are just starting with Python, treat this as a
preview. You can run it and click around without fully
understanding the internals yet.

@author: Hrishikesh Terdalkar
"""

###############################################################################

import io
import os
import json
import base64
import tempfile
from datetime import datetime

import pandas as pd
import numpy as np
import matplotlib

# Use a non-interactive backend so plots work safely in web requests
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from flask import Flask, render_template, request, jsonify, send_file

###############################################################################

# For this demonstration, we'll create a simple Flask app
# In production, you'd separate templates into their own files

app = Flask(__name__)


class ResearchDashboard:
    """Research dashboard backend functionality"""
```

```
    def __init__(self):
        self.experiments = self.load_sample_data()

    def load_sample_data(self):
        """Load sample research data for demonstration"""
        # Generate sample engineering data
        experiments = {}

        # Thermal experiment
        time_points = np.arange(0, 120, 5)
        experiments["thermal_study"] = {
            "time": time_points,
            "temperature": 25
            + 10 * np.sin(2 * np.pi * time_points / 60)
            + np.random.normal(0, 1, len(time_points)),
            "heat_flow": 100
            + 30 * np.cos(2 * np.pi * time_points / 30)
            + np.random.normal(0, 5, len(time_points)),
        }

        # Structural experiment
        load_points = np.linspace(0, 1000, 50)
        experiments["structural_test"] = {
            "load": load_points,
            "displacement": 0.1 * load_points
            + 0.001 * load_points**2
            + np.random.normal(0, 0.5, len(load_points)),
            "stress": load_points / 100,  # Simplified stress calculation
        }

        # Fluid dynamics experiment
        velocity_points = np.linspace(0.1, 5.0, 40)
        experiments["flow_analysis"] = {
            "velocity": velocity_points,
            "pressure_drop": 10 * velocity_points**2
            + np.random.normal(0, 2, len(velocity_points)),
            "reynolds_number": 1000 * velocity_points,
        }

        return experiments

    def analyze_experiment(self, experiment_name, analysis_type):
        """Perform analysis on experiment data"""
        data = self.experiments[experiment_name]
        df = pd.DataFrame(data)

        analysis = {
            "experiment": experiment_name,
            "analysis_type": analysis_type,
            "timestamp": datetime.now().isoformat(),
        }

        if analysis_type == "basic_stats":
            # Basic statistical analysis
            numeric_cols = df.select_dtypes(include=[np.number]).columns
            stats = {}
            for col in numeric_cols:
                stats[col] = {
                    "mean": float(df[col].mean()),
                    "std": float(df[col].std()),
                    "min": float(df[col].min()),
                    "max": float(df[col].max()),
                }
            analysis["statistics"] = stats

        elif analysis_type == "trend_analysis":
            # Trend analysis
            trends = {}
            numeric_cols = df.select_dtypes(include=[np.number]).columns

            # Try to find time or independent variable
            indep_vars = [
                col
                for col in df.columns
```

```
                if col in ["time", "load", "velocity"]
            ]
            indep_var = indep_vars[0] if indep_vars else None

            for col in numeric_cols:
                if col != indep_var:
                    if indep_var:
                        # Linear regression
                        slope, intercept = np.polyfit(
                            df[indep_var], df[col], 1
                        )
                        trends[col] = {
                            "slope": float(slope),
                            "intercept": float(intercept),
                            "correlation": float(df[indep_var].corr(df[col])),
                        }

            analysis["trends"] = trends

        return analysis

    def create_plot(self, experiment_name, x_col, y_col, plot_type="line"):
        """Create visualization plot"""
        data = self.experiments[experiment_name]

        plt.figure(figsize=(10, 6))

        if plot_type == "line":
            plt.plot(
                data[x_col], data[y_col], "bo-", linewidth=2, markersize=4
            )
        elif plot_type == "scatter":
            plt.scatter(data[x_col], data[y_col], alpha=0.7, s=30)

        plt.xlabel(x_col)
        plt.ylabel(y_col)
        plt.title(f"{experiment_name}: {y_col} vs {x_col}")
        plt.grid(True, alpha=0.3)

        # Save plot to bytes for web display
        img_bytes = io.BytesIO()
        plt.savefig(img_bytes, format="png", dpi=300, bbox_inches="tight")
        img_bytes.seek(0)
        plt.close()

        # Convert to base64 for HTML embedding
        img_base64 = base64.b64encode(img_bytes.getvalue()).decode()
        return f"{img_base64}"


# Global dashboard instance
dashboard = ResearchDashboard()


# Flask Routes
@app.route("/")
def index():
    """Main dashboard page"""
    experiments_html_parts = []

    for exp_name, exp_data in dashboard.experiments.items():
        df = pd.DataFrame(exp_data)
        columns = list(df.columns)
        description = f"{exp_name.replace('_', ' ').title()} Data"
        options_html = "".join(
            f"<option value='{col}'>{col}</option>" for col in columns
        )
        experiments_html_parts.append(
            f"""
        <div class="experiment">
            <h3>{exp_name.replace('_', ' ').title()}</h3>
            <p>{description} - {len(df)} data points</p>
            <p>Columns: {', '.join(columns)}</p>
            <button onclick="analyzeExperiment('{exp_name}', 'basic_stats')">Basic Statistics
    </button>
```

```
            <button onclick="analyzeExperiment('{exp_name}', 'trend_analysis')">Trend Analysis</button>
            <div style="margin-top: 10px;">
                <label>
                    X:
                    <select id="xcol-{exp_name}">
                        {options_html}
                    </select>
                </label>
                <label>
                    Y:
                    <select id="ycol-{exp_name}">
                        {options_html}
                    </select>
                </label>
                <button onclick="plotFromSelect('{exp_name}')">
                    Plot
                </button>
            </div>
            <div id="results-{exp_name}"></div>
            <div id="plot-{exp_name}"></div>
        </div>
        """
        )

    experiments_html = "\n".join(experiments_html_parts)

    return f"""
    <html>
    <head>
        <title>Research Dashboard</title>
        <style>
            body {{ font-family: Arial, sans-serif; margin: 40px; }}
            .experiment {{ border: 1px solid #ccc; padding: 20px; margin: 10px; border-radius: 5px; }}
            .plot {{ margin: 20px 0; }}
            button {{ padding: 10px 15px; margin: 5px; cursor: pointer; }}
        </style>
    </head>
    <body>
        <h1>Research Data Dashboard</h1>
        <p>Interactive dashboard for research data analysis and visualization</p>

        <h2>Available Experiments</h2>
        {experiments_html}

        <script>
        function analyzeExperiment(experimentName, analysisType) {{
            fetch('/analyze', {{
                method: 'POST',
                headers: {{ 'Content-Type': 'application/json' }},
                body: JSON.stringify({{
                    experiment: experimentName,
                    analysis_type: analysisType
                }})
            }})
            .then(response => response.json())
            .then(data => {{
                document.getElementById('results-' + experimentName).innerHTML =
                    '<h4>Analysis Results:</h4><pre>' + JSON.stringify(data, null, 2) + '</pre>';
            }});
        }}

        function plotFromSelect(experimentName) {{
            const xSelect = document.getElementById('xcol-' + experimentName);
            const ySelect = document.getElementById('ycol-' + experimentName);
            if (!xSelect || !ySelect) return;
            const xCol = xSelect.value;
            const yCol = ySelect.value;
            createPlot(experimentName, xCol, yCol);
        }}

        function createPlot(experimentName, xCol, yCol) {{
            fetch('/plot', {{
```

```
                method: 'POST',
                headers: {{ 'Content-Type': 'application/json' }},
                body: JSON.stringify({{
                    experiment: experimentName,
                    x_column: xCol,
                    y_column: yCol
                }})
            }})
            .then(response => response.json())
            .then(data => {{
                document.getElementById('plot-' + experimentName).innerHTML =
                    '<h4>Plot:</h4><img src="' + data.plot_url + '" style="max-width:
    100%;">';
            }});
        }}
        </script>
    </body>
    </html>
    """


@app.route("/analyze", methods=["POST"])
def analyze_data():
    """API endpoint for data analysis"""
    request_data = request.json
    experiment = request_data.get("experiment")
    analysis_type = request_data.get("analysis_type")

    if experiment not in dashboard.experiments:
        return jsonify({"error": "Experiment not found"}), 404

    analysis_results = dashboard.analyze_experiment(experiment, analysis_type)
    return jsonify(analysis_results)


@app.route("/plot", methods=["POST"])
def create_plot():
    """API endpoint for creating plots"""
    request_data = request.json
    experiment = request_data.get("experiment")
    x_column = request_data.get("x_column")
    y_column = request_data.get("y_column")

    if experiment not in dashboard.experiments:
        return jsonify({"error": "Experiment not found"}), 404

    if (
        x_column not in dashboard.experiments[experiment]
        or y_column not in dashboard.experiments[experiment]
    ):
        return jsonify({"error": "Invalid columns"}), 400

    plot_url = dashboard.create_plot(experiment, x_column, y_column)
    return jsonify({"plot_url": plot_url})


@app.route("/export/<experiment_name>")
def export_data(experiment_name):
    """Export experiment data as CSV"""
    if experiment_name not in dashboard.experiments:
        return "Experiment not found", 404

    data = dashboard.experiments[experiment_name]
    df = pd.DataFrame(data)

    # Create temporary file
    temp_file = tempfile.NamedTemporaryFile(delete=False, suffix=".csv")
    df.to_csv(temp_file.name, index=False)
    temp_file.close()

    return send_file(
        temp_file.name,
        as_attachment=True,
        download_name=f"{experiment_name}_data.csv",
    )
```

```python


@app.route("/api/experiments")
def get_experiments_list():
    """API endpoint to get list of experiments"""
    experiments_list = []

    for exp_name, exp_data in dashboard.experiments.items():
        df = pd.DataFrame(exp_data)
        experiments_list.append(
            {
                "name": exp_name,
                "columns": list(df.columns),
                "record_count": len(df),
                "description": f"{exp_name.replace('_', ' ').title()} Dataset",
            }
        )

    return jsonify(experiments_list)


def run_dashboard_demonstration():
    """Demonstrate dashboard functionality without running Flask server"""
    print("=== RESEARCH DASHBOARD DEMONSTRATION ===")

    # Test analysis functionality
    print("1. Data Analysis Examples:")

    analysis_types = ["basic_stats", "trend_analysis"]
    experiments = ["thermal_study", "structural_test"]

    for exp in experiments:
        for analysis_type in analysis_types:
            results = dashboard.analyze_experiment(exp, analysis_type)
            print(f"\n{exp} - {analysis_type}:")

            if "statistics" in results:
                print("  Statistics calculated for all numeric columns")
            if "trends" in results:
                print(
                    f"  Trends analyzed: {len(results['trends'])} relationships"
                )

    # Test plot generation
    print("\n2. Plot Generation:")
    plot_combinations = [
        ("thermal_study", "time", "temperature"),
        ("structural_test", "load", "displacement"),
        ("flow_analysis", "velocity", "pressure_drop"),
    ]

    for exp, x, y in plot_combinations:
        try:
            plot_data = dashboard.create_plot(exp, x, y)
            print(f"  {exp}: {y} vs {x} - Plot generated successfully")
        except Exception as e:
            print(f"  {exp}: Error generating plot - {e}")

    print("\n3. Dashboard Features:")
    print("  - Interactive web interface")
    print("  - Real-time data analysis")
    print("  - Dynamic plot generation")
    print("  - Data export functionality")
    print("  - REST API for programmatic access")

    print("\nTo run the actual web dashboard:")
    print("  flask --app session4/04_research_dashboard.py run --port 5000")
    print("Then visit http://localhost:5000 in your browser")


if __name__ == "__main__":
    # When run directly, demonstrate functionality
    run_dashboard_demonstration()

    # Uncomment the line below to actually run the Flask server
```

```
    # app.run(debug=True, port=5000)
```

Listing 4: Research dashboard with Flask

session4/05_integration_demo.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Simplified Integration Demo (Session 4)

One script that ties together the core ideas:
- generate a small dataset
- save to CSV and SQLite (no ORM)
- mock an API pull
- run a few parallel CPU jobs
- emit plots and a JSON summary
"""

################################################################################

import argparse
import json
import sqlite3
import time
from pathlib import Path
from typing import Dict, Any

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from concurrent.futures import ProcessPoolExecutor

################################################################################

RNG_SEED = 42
np.random.seed(RNG_SEED)

################################################################################


def create_dataset(rows: int = 60) -> pd.DataFrame:
    """Create a small engineering-style dataset"""
    minutes = np.arange(rows)
    temperature = 24 + 4 * np.sin(minutes / 8) + np.random.normal(0, 0.6, rows)
    pressure = 101.3 + 0.8 * np.cos(minutes / 6) + np.random.normal(0, 0.1, rows)
    flow = 5 + 0.7 * np.sin(minutes / 10) + np.random.normal(0, 0.05, rows)

    return pd.DataFrame(
        {
            "minute": minutes,
            "temperature_c": temperature,
            "pressure_kpa": pressure,
            "flow_l_min": flow,
        }
    )


def save_csv(df: pd.DataFrame, path: Path) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    df.to_csv(path, index=False)
    print(f"[CSV] Saved {len(df)} rows to {path}")


def save_sqlite(df: pd.DataFrame, db_path: Path) -> None:
    db_path.parent.mkdir(parents=True, exist_ok=True)
    with sqlite3.connect(db_path) as conn:
        df.to_sql("measurements", conn, if_exists="replace", index=False)
        count = conn.execute("SELECT COUNT(*) FROM measurements").fetchone()[0]
    print(f"[DB] Wrote {count} rows to measurements in {db_path}")

```

```python

def mock_api_data(days: int = 5) -> pd.DataFrame:
    """Fake external data to avoid network calls"""
    records = []
    for day in range(days):
        records.append(
            {
                "day": day,
                "city": "SampleCity",
                "temperature_c": 18 + np.random.normal(0, 2),
                "humidity_pct": 60 + np.random.normal(0, 8),
                "condition": np.random.choice(["Clear", "Cloudy", "Rain"]),
            }
        )
    df = pd.DataFrame(records)
    print(f"[API] Generated {len(df)} mock weather records")
    return df


def parallel_job(args: tuple[int, int]) -> float:
    """Worker function for parallel jobs (top-level for pickling)"""
    seed, samples = args
    rng = np.random.default_rng(seed)
    data = rng.normal(loc=0.0, scale=1.0, size=samples)
    return float(np.mean(data))


def run_parallel_jobs(num_jobs: int = 6, samples: int = 50_000):
    """Simple CPU-bound jobs to demonstrate multiprocessing"""
    start = time.time()
    args = [(RNG_SEED + i, samples) for i in range(num_jobs)]
    with ProcessPoolExecutor() as pool:
        results = list(pool.map(parallel_job, args))
    elapsed = time.time() - start
    print(f"[PARALLEL] {num_jobs} jobs finished in {elapsed:.2f}s")
    return results, elapsed


def make_plots(measure_df: pd.DataFrame, api_df: pd.DataFrame, out_dir: Path):
    """Create a couple of simple plots"""
    out_dir.mkdir(parents=True, exist_ok=True)
    plt.switch_backend("Agg")

    # Temperature over time
    plt.figure(figsize=(6, 4))
    plt.plot(measure_df["minute"], measure_df["temperature_c"], label="Temp (C)")
    plt.xlabel("Minute")
    plt.ylabel("Temperature (C)")
    plt.title("Temperature Over Time")
    plt.grid(alpha=0.3)
    plt.legend()
    temp_path = out_dir / "temperature_line.png"
    plt.savefig(temp_path, dpi=200, bbox_inches="tight")
    plt.close()

    # Flow vs Pressure
    plt.figure(figsize=(6, 4))
    plt.scatter(measure_df["pressure_kpa"], measure_df["flow_l_min"], alpha=0.7)
    plt.xlabel("Pressure (kPa)")
    plt.ylabel("Flow (L/min)")
    plt.title("Flow vs Pressure")
    plt.grid(alpha=0.3)
    scatter_path = out_dir / "flow_vs_pressure.png"
    plt.savefig(scatter_path, dpi=200, bbox_inches="tight")
    plt.close()

    # Humidity histogram from API-like data
    plt.figure(figsize=(6, 4))
    plt.hist(api_df["humidity_pct"], bins=8, color="#3b82f6", alpha=0.8)
    plt.xlabel("Humidity (%)")
    plt.ylabel("Frequency")
    plt.title("Humidity Distribution")
    plt.grid(alpha=0.3)
    hist_path = out_dir / "humidity_hist.png"
```

```
    plt.savefig(hist_path, dpi=200, bbox_inches="tight")
    plt.close()

    print(f"[PLOTS] Saved plots to {out_dir}")


def build_summary(
    csv_path: Path,
    db_path: Path,
    api_records: int,
    parallel_time: float,
    output_dir: Path,
) -> Dict[str, Any]:
    return {
        "generated_at": time.strftime("%Y-%m-%dT%H:%M:%S"),
        "csv_file": str(csv_path),
        "sqlite_db": str(db_path),
        "api_records": api_records,
        "parallel_time_sec": parallel_time,
        "output_dir": str(output_dir),
    }


def parse_args():
    parser = argparse.ArgumentParser(
        description="Simplified integration demo (CSV + SQLite + mock API + parallel + plots)"
    ,
        epilog="Example: python session4/05_integration_demo.py --output integrated_output",
    )
    parser.add_argument(
        "--output",
        "-o",
        default="integrated_output",
        help="Folder for CSV, DB, plots, and summary JSON",
    )
    parser.add_argument(
        "--rows", "-r", type=int, default=60, help="Number of rows to generate"
    )
    parser.add_argument(
        "--api-days", type=int, default=5, help="Mock API days to generate"
    )
    parser.add_argument(
        "--jobs", type=int, default=6, help="Parallel jobs to run"
    )
    return parser.parse_args()


def main():
    args = parse_args()
    output_dir = Path(args.output)
    data_dir = output_dir / "data"
    plots_dir = output_dir / "plots"
    output_dir.mkdir(parents=True, exist_ok=True)

    print("=== SIMPLIFIED INTEGRATION DEMO ===")
    print(f"Output directory: {output_dir.resolve()}")

    # 1) Dataset -> CSV + SQLite
    df = create_dataset(rows=args.rows)
    csv_path = data_dir / "engineering_test_data.csv"
    db_path = data_dir / "research_data.db"
    save_csv(df, csv_path)
    save_sqlite(df, db_path)

    # 2) Mock API
    api_df = mock_api_data(days=args.api_days)

    # 3) Parallel jobs
    _, elapsed = run_parallel_jobs(num_jobs=args.jobs)

    # 4) Plots
    make_plots(df, api_df, plots_dir)

    # 5) Summary
```

```
    summary = build_summary(
        csv_path=csv_path,
        db_path=db_path,
        api_records=len(api_df),
        parallel_time=elapsed,
        output_dir=output_dir,
    )
    summary_path = output_dir / "integrated_system_report.json"
    with summary_path.open("w", encoding="utf-8") as f:
        json.dump(summary, f, indent=2)
    print(f"[SUMMARY] Saved {summary_path}")
    print("Done.")


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

Listing 5: Integration demo: DB + APIs + parallel