

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.

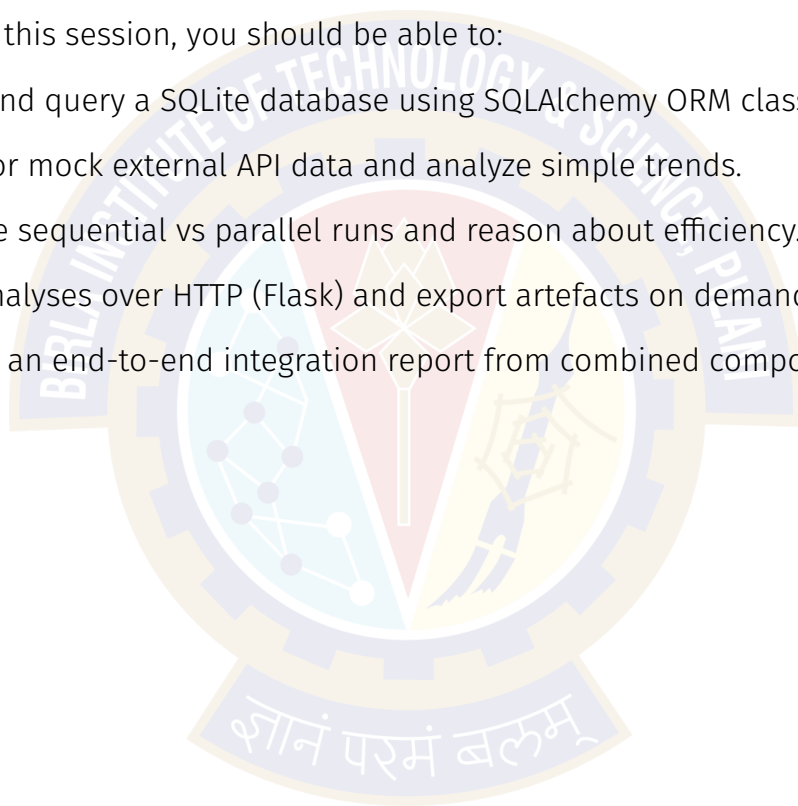
Skills You'll Practice

- 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`.

```
1 # Install dependencies into your active environment
2 pip install -r requirements.txt
3
4 # Prepare shared data produced in Session 3
5 make data
```

How to Run the Examples

1. Persist CSV data to SQLite:

```
1 python session4/01_database.py
2 # Creates research_data.db and writes statistics + CSV export
3
```

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

```
1 python session4/02_api_integration.py
2 # Produces weather_analysis_data.csv for later use
3
```

3. Compare sequential vs parallel workloads:

```
1 python session4/03_parallel_processing.py
2 # Observe speedup and verify identical results across modes
3
```

4. Explore the dashboard (demo or server):

```
1 python session4/04_research_dashboard.py
2 # or run a local server
3 flask -{}-app session4/04_research_dashboard.py run -{}-port 5000
4
```

5. Tie everything together and generate a report:

```
1 python session4/05_integration_demo.py  
2 # Writes integrated_system_report.json  
3
```

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:

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

Simple parallel parameter sweep skeleton:

```
1 from concurrent.futures import ProcessPoolExecutor
2 def simulate(param):
3     # compute something CPU-bound
4     return param, param**2
5 with ProcessPoolExecutor() as ex:
6     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/o1_database.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Research Database Integration (advanced / optional)
5
6  This script uses SQLAlchemy (an Object-Relational Mapper)
7  to talk to a SQLite database. It is here to show what a
8  more 'real world' database layer can look like.
9
10 If you are new to Python, you do not need to understand
11 every line - you can simply run the script once to see
12 that experiments and data points are stored in a database.
13
14 @author: Hrishikesh Terdalkar
15 """
16
17 #####
18
19 import json
20 from datetime import datetime
21 from pathlib import Path
22
23 import numpy as np
24 import pandas as pd
25 from sqlalchemy import (
26     create_engine,
27     Column,
28     Integer,
29     String,
30     Float,
31     DateTime,
32     Text,
33 )
34 from sqlalchemy.ext.declarative import declarative_base
35 from sqlalchemy.orm import sessionmaker
36
37 #####
38
39 RNG_SEED = 42
40 np.random.seed(RNG_SEED)
41
42 Base = declarative_base()
43
44
45 class ResearchExperiment(Base):
46     """ORM class for research experiments"""
47
48     __tablename__ = "research_experiments"
49
50     id = Column(Integer, primary_key=True)
51     experiment_id = Column(String(100), unique=True, nullable=False)
52     title = Column(String(200), nullable=False)
53     description = Column(Text)
54     researcher = Column(String(100))
55     start_date = Column(DateTime)
56     end_date = Column(DateTime)
57     created_at = Column(DateTime, default=datetime.now)
58
59     def __repr__(self):
60         return f"<Experiment(id={self.experiment_id}, title='{self.title}')>"
61
62
63 class ExperimentalData(Base):
64     """ORM class for experimental data points"""
65
66     __tablename__ = "experimental_data"
67
68     id = Column(Integer, primary_key=True)
69     experiment_id = Column(String(100), nullable=False)
```

```

70     timestamp = Column(DateTime, nullable=False)
71     parameter_name = Column(String(100), nullable=False)
72     parameter_value = Column(Float, nullable=False)
73     uncertainty = Column(Float)
74     units = Column(String(50))
75     notes = Column(Text)
76
77     def __repr__(self):
78         return f"<DataPoint(exp={self.experiment_id}, param={self.parameter_name}, value={self
79         .parameter_value})>"
80
81 class ResearchDatabase:
82     """Database management class for research data"""
83
84     def __init__(self, db_url="sqlite:///research_data.db"):
85         self.engine = create_engine(db_url)
86         Base.metadata.create_all(self.engine)
87         Session = sessionmaker(bind=self.engine)
88         self.session = Session()
89
90     def create_experiment(
91         self,
92         experiment_id,
93         title,
94         researcher,
95         description="",
96         start_date=None,
97         end_date=None,
98     ):
99         """Create a new experiment record"""
100         experiment = ResearchExperiment(
101             experiment_id=experiment_id,
102             title=title,
103             researcher=researcher,
104             description=description,
105             start_date=start_date or datetime.now(),
106             end_date=end_date,
107         )
108         self.session.add(experiment)
109         self.session.commit()
110         return experiment
111
112     def add_data_point(
113         self,
114         experiment_id,
115         parameter_name,
116         parameter_value,
117         timestamp=None,
118         uncertainty=None,
119         units="",
120         notes="",
121     ):
122         """Add a single data point"""
123         data_point = ExperimentalData(
124             experiment_id=experiment_id,
125             parameter_name=parameter_name,
126             parameter_value=parameter_value,
127             timestamp=timestamp or datetime.now(),
128             uncertainty=uncertainty,
129             units=units,
130             notes=notes,
131         )
132         self.session.add(data_point)
133         self.session.commit()
134         return data_point
135
136     def add_data_batch(self, experiment_id, data_frame, time_column="time"):
137         """Add multiple data points from a DataFrame"""
138         data_points = []
139
140         for idx, row in data_frame.iterrows():
141             if time_column in data_frame.columns:
142                 timestamp = pd.to_datetime(row[time_column])
143                 if isinstance(timestamp, pd.Timestamp):

```

```
144         timestamp = timestamp.to_pydatetime()
145     else:
146         timestamp = datetime.now()
147
148     for col in data_frame.columns:
149         if col != time_column and pd.api.types.is_numeric_dtype(
150             data_frame[col]
151         ):
152             data_point = ExperimentalData(
153                 experiment_id=experiment_id,
154                 parameter_name=col,
155                 parameter_value=float(row[col]),
156                 timestamp=timestamp,
157             )
158             data_points.append(data_point)
159
160     self.session.bulk_save_objects(data_points)
161     self.session.commit()
162     return len(data_points)
163
164 def get_experiment_data(self, experiment_id, parameter_name=None):
165     """Retrieve data for a specific experiment"""
166     query = self.session.query(ExperimentalData).filter_by(
167         experiment_id=experiment_id
168     )
169
170     if parameter_name:
171         query = query.filter_by(parameter_name=parameter_name)
172
173     results = query.all()
174
175     # Convert to DataFrame for analysis
176     if results:
177         data = []
178         for result in results:
179             data.append(
180                 {
181                     "timestamp": result.timestamp,
182                     "parameter": result.parameter_name,
183                     "value": result.parameter_value,
184                     "uncertainty": result.uncertainty,
185                     "units": result.units,
186                 }
187             )
188         return pd.DataFrame(data)
189     else:
190         return pd.DataFrame()
191
192 def get_experiment_statistics(self, experiment_id):
193     """Calculate statistics for an experiment"""
194     data = self.get_experiment_data(experiment_id)
195
196     if data.empty:
197         return None
198
199     statistics = {}
200     parameters = data["parameter"].unique()
201
202     for param in parameters:
203         param_data = data[data["parameter"] == param][["value"]]
204         statistics[param] = {
205             "count": len(param_data),
206             "mean": float(param_data.mean()),
207             "std": float(param_data.std()),
208             "min": float(param_data.min()),
209             "max": float(param_data.max()),
210         }
211
212     return statistics
213
214 def export_experiment_to_csv(self, experiment_id, output_file):
215     """Export experiment data to CSV"""
216     data = self.get_experiment_data(experiment_id)
217
218     if not data.empty:
```

```
219         data.to_csv(output_file, index=False)
220         return True
221     return False
222
223     def list_experiments(self):
224         """List all experiments in the database"""
225         return self.session.query(ResearchExperiment).all()
226
227
228     def demonstrate_database_operations():
229         """Demonstrate database operations with sample data"""
230         print("== RESEARCH DATABASE DEMONSTRATION ==")
231
232         # Initialize database
233         db = ResearchDatabase()
234
235         # Create sample experiment
236         experiment = db.create_experiment(
237             experiment_id="thermal_study_001",
238             title="Thermal Conductivity Measurement",
239             researcher="PhD Student",
240             description="Measuring thermal conductivity of composite materials",
241         )
242         print(f"Created experiment: {experiment}")
243
244         csv_path = Path("engineering_test_data.csv")
245         if csv_path.exists():
246             thermal_df = pd.read_csv(csv_path)
247             if "time" not in thermal_df.columns:
248                 anchor = pd.Timestamp("2024-01-01 09:00:00")
249                 if "Time_min" in thermal_df.columns:
250                     thermal_df["time"] = anchor + pd.to_timedelta(
251                         thermal_df["Time_min"], unit="m"
252                     )
253             else:
254                 thermal_df["time"] = pd.date_range(
255                     start=anchor, periods=len(thermal_df), freq="min"
256                 )
257             print(f"Loaded dataset from {csv_path}")
258         else:
259             time_points = pd.date_range(start="2024-01-01", periods=100, freq="H")
260             temperatures = (
261                 25
262                 + 10 * np.sin(2 * np.pi * np.arange(100) / 24)
263                 + np.random.normal(0, 1, 100)
264             )
265             heat_flux = (
266                 100
267                 + 20 * np.cos(2 * np.pi * np.arange(100) / 12)
268                 + np.random.normal(0, 5, 100)
269             )
270
271             thermal_df = pd.DataFrame(
272                 {
273                     "time": time_points,
274                     "temperature": temperatures,
275                     "heat_flux": heat_flux,
276                 }
277             )
278             print("Generated synthetic thermal dataset for demonstration")
279
280             # Add data batch
281             points_added = db.add_data_batch("thermal_study_001", thermal_df, "time")
282             print(f"Added {points_added} data points")
283
284             # Retrieve and analyze data
285             data = db.get_experiment_data("thermal_study_001")
286             print(f"Retrieved data shape: {data.shape}")
287
288             statistics = db.get_experiment_statistics("thermal_study_001")
289             print("\nExperiment Statistics:")
290             for param, stats in statistics.items():
291                 print(f"{param}: mean={stats['mean']:.2f} +/- {stats['std']:.2f}")
292
```

```

293     # List all experiments
294     experiments = db.list_experiments()
295     print(f"\nTotal experiments in database: {len(experiments)}")
296
297     # Export to CSV
298     db.export_experiment_to_csv("thermal_study_001", "thermal_data_export.csv")
299     print("Data exported to thermal_data_export.csv")
300
301
302 if __name__ == "__main__":
303     demonstrate_database_operations()

```

Listing 1: SQLAlchemy database integration

session4/o2_api_integration.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  API Integration for Research Data (intermediate)
5
6  This file shows a small, structured way to:
7  - call a free weather API (Open-Meteo) when network access is available
8  - fall back to local mock data if the request fails
9  - turn JSON-like responses into pandas DataFrames
10 - do a bit of trend analysis.
11
12 It also includes a small material-properties client that uses only
13 local mock data, plus a template for APIs that expect bearer tokens.
14
15 Treat the helper classes as ready-made tools - the main focus is the
16 high-level flow in 'demonstrate_api_integration()'.
17
18 @author: Hrishikesh Terdalkar
19 """
20
21 #####
22
23 import time
24 import json
25 from datetime import datetime, timedelta
26
27 import requests
28 import pandas as pd
29 import numpy as np
30
31 #####
32
33 RNG_SEED = 42
34 np.random.seed(RNG_SEED)
35
36
37 class ResearchDataAPI:
38     """Base class for research data API integration"""
39
40     def __init__(self, base_url=None, api_key=None):
41         self.base_url = base_url
42         self.api_key = api_key
43         self.session = requests.Session()
44
45         # Common headers for API requests
46         self.headers = {
47             "User-Agent": "Research Data Collector/1.0",
48             "Accept": "application/json",
49         }
50
51         if api_key:
52             self.headers["Authorization"] = f"Bearer {api_key}"
53
54     def make_request(self, endpoint, params=None, method="GET"):
55         """Make API request with error handling"""

```

```

56     url = f"{self.base_url}/{endpoint}" if self.base_url else endpoint
57
58     try:
59         if method == "GET":
60             response = self.session.get(
61                 url, params=params, headers=self.headers
62             )
63         elif method == "POST":
64             response = self.session.post(
65                 url, json=params, headers=self.headers
66             )
67         else:
68             raise ValueError(f"Unsupported method: {method}")
69
70     response.raise_for_status() # Raise exception for bad status codes
71     return response.json()
72
73 except requests.exceptions.RequestException as e:
74     print(f"API request failed: {e}")
75     return None
76
77 def rate_limit_delay(self, delay_seconds=1):
78     """Simple rate limiting"""
79     time.sleep(delay_seconds)
80
81
82 class WeatherDataCollector(ResearchDataAPI):
83     """Collect weather data for environmental studies"""
84
85     def __init__(self, api_key=None):
86         # Use the free Open-Meteo API for live data when possible.
87         # Documentation: https://open-meteo.com/en/docs
88         super().__init__("https://api.open-meteo.com/v1", api_key)
89
90     @staticmethod
91     def _city_to_coordinates(city, country_code=None):
92         """Map a few example cities to coordinates required by Open-Meteo"""
93         key = f"{city},{country_code}".lower() if country_code else city.lower()
94         mapping = {
95             "london,uk": (51.5074, -0.1278),
96             "london": (51.5074, -0.1278),
97             "mumbai,in": (19.0760, 72.8777),
98             "mumbai": (19.0760, 72.8777),
99             "hyderabad,in": (17.3850, 78.4867),
100            "hyderabad": (17.3850, 78.4867),
101        }
102        return mapping.get(key, (51.5074, -0.1278)) # default: London
103
104     def get_current_weather(self, city, country_code=None):
105         """Get current weather data using Open-Meteo, with mock fallback"""
106         lat, lon = self._city_to_coordinates(city, country_code)
107
108         params = {
109             "latitude": lat,
110             "longitude": lon,
111             "current_weather": "true",
112         }
113
114         raw = self.make_request("forecast", params=params, method="GET")
115
116         if raw and "current_weather" in raw:
117             cw = raw["current_weather"]
118             processed = {
119                 "city": city,
120                 "temperature": cw.get("temperature"),
121                 "pressure": None, # Open-Meteo current_weather does not include pressure
122                 "humidity": None,
123                 "wind_speed": cw.get("windspeed"),
124                 "conditions": "N/A",
125                 "timestamp": datetime.fromisoformat(
126                     cw.get("time")
127                 ).isoformat(),
128             }
129             return processed
130

```

```

131     # Fallback: local mock if API is unreachable or response incomplete
132     mock_data = {
133         "weather": [{"main": "Clear", "description": "clear sky"}],
134         "main": {
135             "temp": 15.5 + np.random.normal(0, 3),
136             "pressure": 1013 + np.random.normal(0, 5),
137             "humidity": 65 + np.random.normal(0, 10),
138             "temp_min": 13.0,
139             "temp_max": 18.0,
140         },
141         "wind": {"speed": 3.1 + np.random.normal(0, 1), "deg": 240},
142         "name": city,
143         "dt": int(datetime.now().timestamp()),
144     }
145
146     return self._process_weather_data(mock_data)
147
148     def get_historical_weather(self, city, days=7):
149         """Generate historical weather data (mock)"""
150         historical_data = []
151         end_date = datetime.now()
152
153         for i in range(days):
154             current_date = end_date - timedelta(days=i)
155
156             # Generate realistic seasonal data
157             day_of_year = current_date.timetuple().tm_yday
158             base_temp = 10 + 10 * np.sin(
159                 2 * np.pi * (day_of_year - 80) / 365
160             ) # Seasonal variation
161
162             daily_data = {
163                 "date": current_date.strftime("%Y-%m-%d"),
164                 "temperature": base_temp + np.random.normal(0, 2),
165                 "pressure": 1013 + np.random.normal(0, 3),
166                 "humidity": 60 + np.random.normal(0, 15),
167                 "wind_speed": 3 + abs(np.random.normal(0, 1.5)),
168                 "conditions": np.random.choice(
169                     ["Clear", "Cloudy", "Rain", "Snow"]
170                 ),
171             }
172             historical_data.append(daily_data)
173
174         return historical_data
175
176     def _process_weather_data(self, raw_data):
177         """Process raw API response into structured format"""
178         return {
179             "city": raw_data.get("name", "Unknown"),
180             "temperature": raw_data["main"]["temp"],
181             "pressure": raw_data["main"]["pressure"],
182             "humidity": raw_data["main"]["humidity"],
183             "wind_speed": raw_data["wind"]["speed"],
184             "conditions": raw_data["weather"][0]["main"],
185             "timestamp": datetime.fromtimestamp(raw_data["dt"]).isoformat(),
186         }
187
188     def analyze_weather_trends(self, historical_data):
189         """Analyze weather trends for research"""
190         df = pd.DataFrame(historical_data)
191         df["date"] = pd.to_datetime(df["date"])
192
193         analysis = {
194             "analysis_period": {
195                 "start": df["date"].min().strftime("%Y-%m-%d"),
196                 "end": df["date"].max().strftime("%Y-%m-%d"),
197                 "days": len(df),
198             },
199             "temperature_analysis": {
200                 "mean": df["temperature"].mean(),
201                 "trend": (
202                     "increasing"
203                     if df["temperature"].iloc[-1] > df["temperature"].iloc[0]
204                     else "decreasing"

```

```
205         ),
206         "daily_variation": df["temperature"].std(),
207     },
208     "pressure_analysis": {
209         "mean": df["pressure"].mean(),
210         "correlation_with_temp": df["temperature"].corr(
211             df["pressure"]
212         ),
213     },
214     "condition_frequency": df["conditions"].value_counts().to_dict(),
215 }
216
217 return analysis, df
218
219
220 class MaterialPropertiesAPI(ResearchDataAPI):
221     """Mock API for material properties data"""
222
223     def __init__(self):
224         # Uses only local mock data; URL and key are placeholders.
225         super().__init__("https://example.com/materials-api", "demo_key")
226
227         # Mock material database
228         self.materials_db = {
229             "aluminum": {
230                 "density": 2.70, # g/cm^3
231                 "youngs_modulus": 69, # GPa
232                 "thermal_conductivity": 237, # W/m*K
233                 "specific_heat": 0.897, # J/g*K
234             },
235             "steel": {
236                 "density": 7.85,
237                 "youngs_modulus": 200,
238                 "thermal_conductivity": 50,
239                 "specific_heat": 0.466,
240             },
241             "copper": {
242                 "density": 8.96,
243                 "youngs_modulus": 110,
244                 "thermal_conductivity": 401,
245                 "specific_heat": 0.385,
246             },
247         }
248
249     def get_material_properties(self, material_name):
250         """Get properties for a specific material"""
251         material_name = material_name.lower()
252
253         if material_name in self.materials_db:
254             return self.materials_db[material_name]
255         else:
256             return {"error": f"Material '{material_name}' not found"}
257
258     def compare_materials(self, material_list, property_name):
259         """Compare specific property across materials"""
260         comparison = {}
261
262         for material in material_list:
263             props = self.get_material_properties(material)
264             if property_name in props:
265                 comparison[material] = props[property_name]
266
267         return comparison
268
269
270 def demonstrate_api_integration():
271     """Demonstrate API integration with research data"""
272     print("=== API INTEGRATION DEMONSTRATION ===")
273
274     # Weather data collection
275     weather_collector = WeatherDataCollector()
276
277     print("1. Current Weather Data:")
278     current_weather = weather_collector.get_current_weather("London", "UK")
279     for key, value in current_weather.items():
```

```

280     print(f"    {key}: {value}")
281
282     print("\n2. Historical Weather Analysis:")
283     historical_data = weather_collector.get_historical_weather("London", 30)
284     analysis, weather_df = weather_collector.analyze_weather_trends(
285         historical_data
286     )
287
288     print(
289         f"    Period: {analysis['analysis_period']['start']} to {analysis['analysis_period']['end']}"
290     )
291     print(
292         f"    Mean temperature: {analysis['temperature_analysis']['mean']:.1f} deg C"
293     )
294     print(f"    Trend: {analysis['temperature_analysis']['trend']}")
295
296     # Material properties API
297     materials_api = MaterialPropertiesAPI()
298
299     print("\n3. Material Properties:")
300     materials = ["aluminum", "steel", "copper"]
301     for material in materials:
302         props = materials_api.get_material_properties(material)
303         print(f"    {material.capitalize()}:")
304         print(f"        Density: {props['density']} g/cm^3")
305         print(f"        Young's Modulus: {props['youngs_modulus']} GPa")
306
307     print("\n4. Material Comparison (Thermal Conductivity):")
308     comparison = materials_api.compare_materials(
309         materials, "thermal_conductivity"
310     )
311     for material, conductivity in comparison.items():
312         print(f"    {material}: {conductivity} W/m*K")
313
314     # Save data for further analysis
315     weather_df.to_csv("weather_analysis_data.csv", index=False)
316     print("\nWeather data saved to weather_analysis_data.csv")
317
318 if __name__ == "__main__":
319     demonstrate_api_integration()
320

```

Listing 2: API integration (mock weather and materials)

session4/o3_parallel_processing.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Parallel Processing for Research (intermediate)
5
6  Goal: show how running the same calculation many times
7  can be sped up by using multiple CPU cores.
8
9  You do not need to understand every detail of the Monte
10 Carlo examples - focus on the difference between the
11 sequential and parallel timings printed by the script.
12
13 @author: Hrishikesh Terdalkar
14 """
15
16 #####
17
18 import time
19 import multiprocessing as mp
20 from concurrent.futures import ProcessPoolExecutor
21
22 import numpy as np
23 import pandas as pd

```

```
24 #####
25 #####
26
27 RNG_SEED = 42
28 np.random.seed(RNG_SEED)
29
30
31 class ResearchParallelProcessor:
32     """Parallel processing utilities for research applications"""
33
34     @staticmethod
35     def get_system_info():
36         """Get information about available processing resources"""
37         cpu_total = mp.cpu_count()
38         print("=== SYSTEM INFORMATION ===")
39         print(f"CPU Cores: {cpu_total}")
40
41         return cpu_total
42
43
44 class MonteCarloSimulator:
45     """Parallel Monte Carlo simulation for engineering applications"""
46
47     def __init__(self):
48         self.results = []
49
50     @staticmethod
51     def monte_carlo_trial(
52         trial_id, num_samples=1000, simulation_type="structural"
53     ):
54         """Single Monte Carlo trial - simulating different engineering scenarios"""
55
56         if simulation_type == "structural":
57             # Structural reliability simulation
58             load = np.random.normal(100, 15, num_samples) # kN - random load
59             strength = np.random.normal(
60                 150, 20, num_samples
61             ) # kN - material strength
62             safety_margin = strength - load
63             failures = np.sum(safety_margin < 0)
64             reliability = 1 - (failures / num_samples)
65
66             return {
67                 "trial_id": trial_id,
68                 "reliability": reliability,
69                 "mean_safety_margin": np.mean(safety_margin),
70                 "failure_probability": failures / num_samples,
71             }
72
73         elif simulation_type == "thermal":
74             # Thermal analysis simulation
75             initial_temp = np.random.normal(20, 2, num_samples)
76             heat_input = np.random.normal(1000, 100, num_samples)
77             material_resistance = np.random.normal(0.5, 0.1, num_samples)
78             final_temp = initial_temp + heat_input * material_resistance
79
80             # Check for overheating
81             overheat_count = np.sum(final_temp > 100)
82
83             return {
84                 "trial_id": trial_id,
85                 "mean_final_temp": np.mean(final_temp),
86                 "overheat_probability": overheat_count / num_samples,
87                 "temp_std": np.std(final_temp),
88             }
89
90         else:
91             raise ValueError(f"Unknown simulation type: {simulation_type}")
92
93     def run_sequential_simulation(
94         self, num_trials=100, num_samples=1000, simulation_type="structural"
95     ):
96         """Run simulation sequentially for comparison"""
97         print(f"Running {num_trials} {simulation_type} trials sequentially...")
98
```

```
99     start_time = time.time()
100     results = []
101
102     for i in range(num_trials):
103         result = self.monte_carlo_trial(i, num_samples, simulation_type)
104         results.append(result)
105
106     sequential_time = time.time() - start_time
107     return results, sequential_time
108
109     def run_parallel_simulation(
110         self,
111         num_trials=100,
112         num_samples=1000,
113         simulation_type="structural",
114         num_workers=None,
115     ):
116         """Run simulation in parallel"""
117         if num_workers is None:
118             num_workers = mp.cpu_count()
119
120         print(
121             f"Running {num_trials} {simulation_type} trials in parallel ({num_workers} workers
122         )..."
123
124         start_time = time.time()
125
126         with ProcessPoolExecutor(max_workers=num_workers) as executor:
127             futures = [
128                 executor.submit(
129                     self.monte_carlo_trial, i, num_samples, simulation_type
130                 )
131                 for i in range(num_trials)
132             ]
133             results = [future.result() for future in futures]
134
135         parallel_time = time.time() - start_time
136         return results, parallel_time
137
138     def analyze_simulation_results(self, results, simulation_type):
139         """Analyze and summarize simulation results"""
140         if simulation_type == "structural":
141             reliabilities = [r["reliability"] for r in results]
142             safety_margins = [r["mean_safety_margin"] for r in results]
143
144             summary = {
145                 "simulation_type": simulation_type,
146                 "trials": len(results),
147                 "mean_reliability": np.mean(reliabilities),
148                 "reliability_std": np.std(reliabilities),
149                 "mean_safety_margin": np.mean(safety_margins),
150                 "reliability_95_ci": [
151                     np.percentile(reliabilities, 2.5),
152                     np.percentile(reliabilities, 97.5),
153                 ],
154             }
155
156         elif simulation_type == "thermal":
157             final_temps = [r["mean_final_temp"] for r in results]
158             overheat_probs = [r["overheat_probability"] for r in results]
159
160             summary = {
161                 "simulation_type": simulation_type,
162                 "trials": len(results),
163                 "mean_final_temp": np.mean(final_temps),
164                 "overheat_probability": np.mean(overheat_probs),
165                 "temp_variation": np.std(final_temps),
166             }
167
168         return summary
169
170     class DataBatchProcessor:
171         """Parallel processing for batch data operations"""
172
```

```

173
174 @staticmethod
175 def process_data_chunk(chunk_data, operation="statistics"):
176     """Process a chunk of data with specified operation"""
177
178     if operation == "statistics":
179         return {
180             "chunk_size": len(chunk_data),
181             "mean": np.mean(chunk_data),
182             "std": np.std(chunk_data),
183             "min": np.min(chunk_data),
184             "max": np.max(chunk_data),
185         }
186
187     elif operation == "fft":
188         # Simulate FFT analysis
189         fft_result = np.fft.fft(chunk_data)
190         magnitude = np.abs(fft_result)
191         return {
192             "chunk_size": len(chunk_data),
193             "dominant_frequency": np.argmax(
194                 magnitude[1 : len(magnitude) // 2]
195             )
196             + 1,
197             "max_magnitude": np.max(magnitude),
198         }
199
200     else:
201         raise ValueError(f"Unknown operation: {operation}")
202
203 def process_large_dataset(
204     self, data, chunk_size=1000, operation="statistics", parallel=True
205 ):
206     """Process large dataset in chunks, optionally in parallel"""
207     chunks = [
208         data[i : i + chunk_size] for i in range(0, len(data), chunk_size)
209     ]
210     print(f"Processing {len(data)} points in {len(chunks)} chunks...")
211
212     start_time = time.time()
213
214     if parallel:
215         with ProcessPoolExecutor() as executor:
216             futures = [
217                 executor.submit(self.process_data_chunk, chunk, operation)
218                 for chunk in chunks
219             ]
220             results = [future.result() for future in futures]
221     else:
222         results = [
223             self.process_data_chunk(chunk, operation) for chunk in chunks
224         ]
225
226     processing_time = time.time() - start_time
227     return results, processing_time
228
229
230 def demonstrate_parallel_processing():
231     """Demonstrate parallel processing capabilities"""
232     print("=== PARALLEL PROCESSING DEMONSTRATION ===")
233
234     # System information
235     cpu_count = ResearchParallelProcessor.get_system_info()
236
237     # Monte Carlo simulation comparison
238     simulator = MonteCarloSimulator()
239
240     print("\n1. Monte Carlo Simulation - Structural Reliability")
241     print("-" * 50)
242
243     # Sequential simulation
244     seq_results, seq_time = simulator.run_sequential_simulation(
245         num_trials=50, num_samples=5000, simulation_type="structural"
246     )
247

```

```
248 # Parallel simulation
249 par_results, par_time = simulator.run_parallel_simulation(
250     num_trials=50, num_samples=5000, simulation_type="structural"
251 )
252
253 # Performance comparison
254 speedup = seq_time / par_time
255 efficiency = (speedup / cpu_count) * 100
256
257 print(f"Sequential time: {seq_time:.2f} seconds")
258 print(f"Parallel time: {par_time:.2f} seconds")
259 print(f"Speedup: {speedup:.2f}x")
260 print(f"Efficiency: {efficiency:.1f}%")
261
262 # Analyze results
263 summary = simulator.analyze_simulation_results(par_results, "structural")
264 print(f"\nSimulation Results:")
265 print(f"Mean reliability: {summary['mean_reliability']:.3f}")
266 print(f"Reliability std: {summary['reliability_std']:.3f}")
267 print(
268     f"95% CI: [{summary['reliability_95_ci'][0]:.3f}, {summary['reliability_95_ci'][1]:.3f}]"
269 )
270
271 # Thermal simulation
272 print("\n2. Monte Carlo Simulation - Thermal Analysis")
273 print("-" * 50)
274
275 thermal_results, thermal_time = simulator.run_parallel_simulation(
276     num_trials=30, simulation_type="thermal"
277 )
278
279 thermal_summary = simulator.analyze_simulation_results(
280     thermal_results, "thermal"
281 )
282 print(f"Parallel time: {thermal_time:.2f} seconds")
283 print(
284     f"Mean final temperature: {thermal_summary['mean_final_temp']:.1f} deg C"
285 )
286 print(
287     f"Overheat probability: {thermal_summary['overheat_probability']:.3f}"
288 )
289
290 # Batch data processing
291 print("\n3. Batch Data Processing")
292 print("-" * 50)
293
294 processor = DataBatchProcessor()
295
296 # Generate large dataset
297 large_data = np.random.normal(0, 1, 50000) # 50,000 data points
298
299 # Sequential processing
300 seq_batch, seq_batch_time = processor.process_large_dataset(
301     large_data, chunk_size=1000, operation="statistics", parallel=False
302 )
303
304 # Parallel processing
305 par_batch, par_batch_time = processor.process_large_dataset(
306     large_data, chunk_size=1000, operation="statistics", parallel=True
307 )
308
309 batch_speedup = seq_batch_time / par_batch_time
310
311 print(f"Dataset size: {len(large_data)} points")
312 print(f"Sequential batch time: {seq_batch_time:.2f} seconds")
313 print(f"Parallel batch time: {par_batch_time:.2f} seconds")
314 print(f"Batch processing speedup: {batch_speedup:.2f}x")
315
316 # Verify results are consistent
317 seq_means = [chunk["mean"] for chunk in seq_batch]
318 par_means = [chunk["mean"] for chunk in par_batch]
319
320 if np.allclose(seq_means, par_means, rtol=1e-10):
```

```

321     print("[OK] Sequential and parallel results are identical")
322 else:
323     print(
324         "[WARN] Results differ between sequential and parallel processing"
325     )
326
327
328 if __name__ == "__main__":
329     demonstrate_parallel_processing()

```

Listing 3: Parallel processing and Monte Carlo

session4/o4_research_dashboard.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Research Dashboard with Flask (advanced / optional)
5
6  This script is a larger example that combines several
7  ideas from the rest of Session 4 into a small web app.
8
9  If you are just starting with Python, treat this as a
10 preview. You can run it and click around without fully
11 understanding the internals yet.
12
13 @author: Hrishikesh Terdalkar
14 """
15
16 #####
17
18 import io
19 import os
20 import json
21 import base64
22 import tempfile
23 from datetime import datetime
24
25 import pandas as pd
26 import numpy as np
27 import matplotlib.pyplot as plt
28 from flask import Flask, render_template, request, jsonify, send_file
29
30 #####
31
32 # For this demonstration, we'll create a simple Flask app
33 # In production, you'd separate templates into their own files
34
35 app = Flask(__name__)
36
37 class ResearchDashboard:
38     """Research dashboard backend functionality"""
39
40     def __init__(self):
41         self.experiments = self.load_sample_data()
42
43     def load_sample_data(self):
44         """Load sample research data for demonstration"""
45         # Generate sample engineering data
46         experiments = {}
47
48         # Thermal experiment
49         time_points = np.arange(0, 120, 5)
50         experiments["thermal_study"] = {
51             "time": time_points,
52             "temperature": 25
53             + 10 * np.sin(2 * np.pi * time_points / 60)
54             + np.random.normal(0, 1, len(time_points)),
55             "heat_flow": 100
56             + 30 * np.cos(2 * np.pi * time_points / 30)

```

```

58         + np.random.normal(0, 5, len(time_points)),
59     }
60
61     # Structural experiment
62     load_points = np.linspace(0, 1000, 50)
63     experiments["structural_test"] = {
64         "load": load_points,
65         "displacement": 0.1 * load_points
66         + 0.001 * load_points**2
67         + np.random.normal(0, 0.5, len(load_points)),
68         "stress": load_points / 100, # Simplified stress calculation
69     }
70
71     # Fluid dynamics experiment
72     velocity_points = np.linspace(0.1, 5.0, 40)
73     experiments["flow_analysis"] = {
74         "velocity": velocity_points,
75         "pressure_drop": 10 * velocity_points**2
76         + np.random.normal(0, 2, len(velocity_points)),
77         "reynolds_number": 1000 * velocity_points,
78     }
79
80     return experiments
81
82     def analyze_experiment(self, experiment_name, analysis_type):
83         """Perform analysis on experiment data"""
84         data = self.experiments[experiment_name]
85         df = pd.DataFrame(data)
86
87         analysis = {
88             "experiment": experiment_name,
89             "analysis_type": analysis_type,
90             "timestamp": datetime.now().isoformat(),
91         }
92
93         if analysis_type == "basic_stats":
94             # Basic statistical analysis
95             numeric_cols = df.select_dtypes(include=[np.number]).columns
96             stats = {}
97             for col in numeric_cols:
98                 stats[col] = {
99                     "mean": float(df[col].mean()),
100                    "std": float(df[col].std()),
101                    "min": float(df[col].min()),
102                    "max": float(df[col].max()),
103                }
104             analysis["statistics"] = stats
105
106         elif analysis_type == "trend_analysis":
107             # Trend analysis
108             trends = {}
109             numeric_cols = df.select_dtypes(include=[np.number]).columns
110
111             # Try to find time or independent variable
112             indep_vars = [
113                 col
114                 for col in df.columns
115                 if col in ["time", "load", "velocity"]
116             ]
117             indep_var = indep_vars[0] if indep_vars else None
118
119             for col in numeric_cols:
120                 if col != indep_var:
121                     if indep_var:
122                         # Linear regression
123                         slope, intercept = np.polyfit(
124                             df[indep_var], df[col], 1
125                         )
126                         trends[col] = {
127                             "slope": float(slope),
128                             "intercept": float(intercept),
129                             "correlation": float(df[indep_var].corr(df[col])),
130                         }
131
132             analysis["trends"] = trends

```

```

133     return analysis
134
135 def create_plot(self, experiment_name, x_col, y_col, plot_type="line"):
136     """Create visualization plot"""
137     data = self.experiments[experiment_name]
138
139     plt.figure(figsize=(10, 6))
140
141     if plot_type == "line":
142         plt.plot(
143             data[x_col], data[y_col], "bo-", linewidth=2, markersize=4
144         )
145     elif plot_type == "scatter":
146         plt.scatter(data[x_col], data[y_col], alpha=0.7, s=30)
147
148     plt.xlabel(x_col)
149     plt.ylabel(y_col)
150     plt.title(f"{experiment_name}: {y_col} vs {x_col}")
151     plt.grid(True, alpha=0.3)
152
153     # Save plot to bytes for web display
154     img_bytes = io.BytesIO()
155     plt.savefig(img_bytes, format="png", dpi=300, bbox_inches="tight")
156     img_bytes.seek(0)
157     plt.close()
158
159     # Convert to base64 for HTML embedding
160     img_base64 = base64.b64encode(img_bytes.getvalue()).decode()
161     return f"data:image/png;base64,{img_base64}"
162
163
164 # Global dashboard instance
165 dashboard = ResearchDashboard()
166
167
168 # Flask Routes
169 @app.route("/")
170 def index():
171     """Main dashboard page"""
172     experiments_info = []
173
174     for exp_name, exp_data in dashboard.experiments.items():
175         df = pd.DataFrame(exp_data)
176         experiments_info.append(
177             {
178                 "name": exp_name,
179                 "columns": list(df.columns),
180                 "data_points": len(df),
181                 "description": f"{exp_name.replace('_', ' ').title()} Data",
182             }
183         )
184
185     return f"""
186     <html>
187     <head>
188         <title>Research Dashboard</title>
189         <style>
190             body {{ font-family: Arial, sans-serif; margin: 40px; }}
191             .experiment {{ border: 1px solid #ccc; padding: 20px; margin: 10px; border-radius:
192     5px; }}
193             .plot {{ margin: 20px 0; }}
194             button {{ padding: 10px 15px; margin: 5px; cursor: pointer; }}
195         </style>
196     </head>
197     <body>
198         <h1>Research Data Dashboard</h1>
199         <p>Interactive dashboard for research data analysis and visualization</p>
200
201         <h2>Available Experiments</h2>
202         { ''.join([f'''
203         <div class="experiment">
204             <h3>{exp['name'].replace('_', ' ').title()}</h3>
205             <p>{exp['description']} - {exp['data_points']} data points</p>
206             <p>Columns: {', '.join(exp['columns'])}</p>

```

```

207         <button onclick="analyzeExperiment('{exp['name']}','basic_stats')">Basic
Statistics</button>
208         <button onclick="analyzeExperiment('{exp['name']}','trend_analysis')">Trend
Analysis</button>
209         <div id="results-{exp['name']}"></div>
210         <div id="plot-{exp['name']}"></div>
211     </div>
212     ''' for exp in experiments_info]]
213
214     <script>
215     function analyzeExperiment(experimentName, analysisType) {{
216         fetch('/analyze', {{
217             method: 'POST',
218             headers: {{ 'Content-Type': 'application/json' }},
219             body: JSON.stringify({{
220                 experiment: experimentName,
221                 analysis_type: analysisType
222             }})
223         })
224         .then(response => response.json())
225         .then(data => {{
226             document.getElementById('results-' + experimentName).innerHTML =
227             '<h4>Analysis Results:</h4><pre>' + JSON.stringify(data, null, 2) + '</pre>
>';
228         }});
229     }}
230
231     function createPlot(experimentName, xCol, yCol) {{
232         fetch('/plot', {{
233             method: 'POST',
234             headers: {{ 'Content-Type': 'application/json' }},
235             body: JSON.stringify({{
236                 experiment: experimentName,
237                 x_column: xCol,
238                 y_column: yCol
239             }})
240         })
241         .then(response => response.json())
242         .then(data => {{
243             document.getElementById('plot-' + experimentName).innerHTML =
244             '<h4>Plot:</h4>';
245         }});
246     }}
247     </script>
248 </body>
249 </html>
250 """
251
252
253 @app.route("/analyze", methods=["POST"])
254 def analyze_data():
255     """API endpoint for data analysis"""
256     request_data = request.json
257     experiment = request_data.get("experiment")
258     analysis_type = request_data.get("analysis_type")
259
260     if experiment not in dashboard.experiments:
261         return jsonify({"error": "Experiment not found"}), 404
262
263     analysis_results = dashboard.analyze_experiment(experiment, analysis_type)
264     return jsonify(analysis_results)
265
266
267 @app.route("/plot", methods=["POST"])
268 def create_plot():
269     """API endpoint for creating plots"""
270     request_data = request.json
271     experiment = request_data.get("experiment")
272     x_column = request_data.get("x_column")
273     y_column = request_data.get("y_column")
274
275     if experiment not in dashboard.experiments:
276         return jsonify({"error": "Experiment not found"}), 404

```

```
277
278     if (
279         x_column not in dashboard.experiments[experiment]
280         or y_column not in dashboard.experiments[experiment]
281     ):
282         return jsonify({"error": "Invalid columns"}), 400
283
284     plot_url = dashboard.create_plot(experiment, x_column, y_column)
285     return jsonify({"plot_url": plot_url})
286
287
288 @app.route("/export/<experiment_name>")
289 def export_data(experiment_name):
290     """Export experiment data as CSV"""
291     if experiment_name not in dashboard.experiments:
292         return "Experiment not found", 404
293
294     data = dashboard.experiments[experiment_name]
295     df = pd.DataFrame(data)
296
297     # Create temporary file
298     temp_file = tempfile.NamedTemporaryFile(delete=False, suffix=".csv")
299     df.to_csv(temp_file.name, index=False)
300     temp_file.close()
301
302     return send_file(
303         temp_file.name,
304         as_attachment=True,
305         download_name=f"{experiment_name}_data.csv",
306     )
307
308
309 @app.route("/api/experiments")
310 def get_experiments_list():
311     """API endpoint to get list of experiments"""
312     experiments_list = []
313
314     for exp_name, exp_data in dashboard.experiments.items():
315         df = pd.DataFrame(exp_data)
316         experiments_list.append(
317             {
318                 "name": exp_name,
319                 "columns": list(df.columns),
320                 "record_count": len(df),
321                 "description": f"{exp_name.replace('_', ' ').title()} Dataset",
322             }
323         )
324
325     return jsonify(experiments_list)
326
327
328 def run_dashboard_demonstration():
329     """Demonstrate dashboard functionality without running Flask server"""
330     print("=== RESEARCH DASHBOARD DEMONSTRATION ===")
331
332     # Test analysis functionality
333     print("1. Data Analysis Examples:")
334
335     analysis_types = ["basic_stats", "trend_analysis"]
336     experiments = ["thermal_study", "structural_test"]
337
338     for exp in experiments:
339         for analysis_type in analysis_types:
340             results = dashboard.analyze_experiment(exp, analysis_type)
341             print(f"\n{exp} - {analysis_type}:")
342
343             if "statistics" in results:
344                 print("    Statistics calculated for all numeric columns")
345             if "trends" in results:
346                 print(
347                     f"    Trends analyzed: {len(results['trends'])} relationships"
348                 )
349
350     # Test plot generation
351     print("\n2. Plot Generation:")
```

```

352 plot_combinations = [
353     ("thermal_study", "time", "temperature"),
354     ("structural_test", "load", "displacement"),
355     ("flow_analysis", "velocity", "pressure_drop"),
356 ]
357
358 for exp, x, y in plot_combinations:
359     try:
360         plot_data = dashboard.create_plot(exp, x, y)
361         print(f" {exp}: {y} vs {x} - Plot generated successfully")
362     except Exception as e:
363         print(f" {exp}: Error generating plot - {e}")
364
365 print("\n3. Dashboard Features:")
366 print(" - Interactive web interface")
367 print(" - Real-time data analysis")
368 print(" - Dynamic plot generation")
369 print(" - Data export functionality")
370 print(" - REST API for programmatic access")
371
372 print("\nTo run the actual web dashboard:")
373 print(" flask --app session4/04_research_dashboard.py run --port 5000")
374 print("Then visit http://localhost:5000 in your browser")
375
376
377 if __name__ == "__main__":
378     # When run directly, demonstrate functionality
379     run_dashboard_demonstration()
380
381     # Uncomment the line below to actually run the Flask server
382     # app.run(debug=True, port=5000)

```

Listing 4: Research dashboard with Flask

session4/05_integration_demo.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Simplified Integration Demo (Session 4)
5
6  One script that ties together the core ideas:
7  - generate a small dataset
8  - save to CSV and SQLite (no ORM)
9  - mock an API pull
10 - run a few parallel CPU jobs
11 - emit plots and a JSON summary
12 """
13
14 #####
15
16 import argparse
17 import json
18 import sqlite3
19 import time
20 from pathlib import Path
21 from typing import Dict, Any
22
23 import numpy as np
24 import pandas as pd
25 import matplotlib.pyplot as plt
26 from concurrent.futures import ProcessPoolExecutor
27
28 #####
29
30 RNG_SEED = 42
31 np.random.seed(RNG_SEED)
32
33 #####
34
35

```

```

36 def create_dataset(rows: int = 60) -> pd.DataFrame:
37     """Create a small engineering-style dataset"""
38     minutes = np.arange(rows)
39     temperature = 24 + 4 * np.sin(minutes / 8) + np.random.normal(0, 0.6, rows)
40     pressure = 101.3 + 0.8 * np.cos(minutes / 6) + np.random.normal(0, 0.1, rows)
41     flow = 5 + 0.7 * np.sin(minutes / 10) + np.random.normal(0, 0.05, rows)
42
43     return pd.DataFrame(
44         {
45             "minute": minutes,
46             "temperature_c": temperature,
47             "pressure_kpa": pressure,
48             "flow_l_min": flow,
49         }
50     )
51
52
53 def save_csv(df: pd.DataFrame, path: Path) -> None:
54     path.parent.mkdir(parents=True, exist_ok=True)
55     df.to_csv(path, index=False)
56     print(f"[CSV] Saved {len(df)} rows to {path}")
57
58
59 def save_sqlite(df: pd.DataFrame, db_path: Path) -> None:
60     db_path.parent.mkdir(parents=True, exist_ok=True)
61     with sqlite3.connect(db_path) as conn:
62         df.to_sql("measurements", conn, if_exists="replace", index=False)
63         count = conn.execute("SELECT COUNT(*) FROM measurements").fetchone()[0]
64         print(f"[DB] Wrote {count} rows to measurements in {db_path}")
65
66
67 def mock_api_data(days: int = 5) -> pd.DataFrame:
68     """Fake external data to avoid network calls"""
69     records = []
70     for day in range(days):
71         records.append(
72             {
73                 "day": day,
74                 "city": "SampleCity",
75                 "temperature_c": 18 + np.random.normal(0, 2),
76                 "humidity_pct": 60 + np.random.normal(0, 8),
77                 "condition": np.random.choice(["Clear", "Cloudy", "Rain"]),
78             }
79         )
80     df = pd.DataFrame(records)
81     print(f"[API] Generated {len(df)} mock weather records")
82     return df
83
84
85 def run_parallel_jobs(num_jobs: int = 6, samples: int = 50_000):
86     """Simple CPU-bound jobs to demonstrate multiprocessing"""
87
88     def job(seed: int) -> float:
89         rng = np.random.default_rng(seed)
90         data = rng.normal(loc=0.0, scale=1.0, size=samples)
91         return float(np.mean(data))
92
93     start = time.time()
94     seeds = [RNG_SEED + i for i in range(num_jobs)]
95     with ProcessPoolExecutor() as pool:
96         results = list(pool.map(job, seeds))
97     elapsed = time.time() - start
98     print(f"[PARALLEL] {num_jobs} jobs finished in {elapsed:.2f}s")
99     return results, elapsed
100
101
102 def make_plots(measure_df: pd.DataFrame, api_df: pd.DataFrame, out_dir: Path):
103     """Create a couple of simple plots"""
104     out_dir.mkdir(parents=True, exist_ok=True)
105     plt.switch_backend("Agg")
106
107     # Temperature over time
108     plt.figure(figsize=(6, 4))
109     plt.plot(measure_df["minute"], measure_df["temperature_c"], label="Temp (C)")

```

```

110 plt.xlabel("Minute")
111 plt.ylabel("Temperature (C)")
112 plt.title("Temperature Over Time")
113 plt.grid(alpha=0.3)
114 plt.legend()
115 temp_path = out_dir / "temperature_line.png"
116 plt.savefig(temp_path, dpi=200, bbox_inches="tight")
117 plt.close()
118
119 # Flow vs Pressure
120 plt.figure(figsize=(6, 4))
121 plt.scatter(measure_df["pressure_kpa"], measure_df["flow_l_min"], alpha=0.7)
122 plt.xlabel("Pressure (kPa)")
123 plt.ylabel("Flow (L/min)")
124 plt.title("Flow vs Pressure")
125 plt.grid(alpha=0.3)
126 scatter_path = out_dir / "flow_vs_pressure.png"
127 plt.savefig(scatter_path, dpi=200, bbox_inches="tight")
128 plt.close()
129
130 # Humidity histogram from API-like data
131 plt.figure(figsize=(6, 4))
132 plt.hist(api_df["humidity_pct"], bins=8, color="#3b82f6", alpha=0.8)
133 plt.xlabel("Humidity (%)")
134 plt.ylabel("Frequency")
135 plt.title("Humidity Distribution")
136 plt.grid(alpha=0.3)
137 hist_path = out_dir / "humidity_hist.png"
138 plt.savefig(hist_path, dpi=200, bbox_inches="tight")
139 plt.close()
140
141 print(f"[PLOTS] Saved plots to {out_dir}")
142
143
144 def build_summary(
145     csv_path: Path,
146     db_path: Path,
147     api_records: int,
148     parallel_time: float,
149     output_dir: Path,
150 ) -> Dict[str, Any]:
151     return {
152         "generated_at": time.strftime("%Y-%m-%dT%H:%M:%S"),
153         "csv_file": str(csv_path),
154         "sqlite_db": str(db_path),
155         "api_records": api_records,
156         "parallel_time_sec": parallel_time,
157         "output_dir": str(output_dir),
158     }
159
160
161 def parse_args():
162     parser = argparse.ArgumentParser(
163         description="Simplified integration demo (CSV + SQLite + mock API + parallel + plots)"
164         ,
165         epilog="Example: python session4/05_integration_demo.py --output integrated_output",
166     )
167     parser.add_argument(
168         "--output",
169         "-o",
170         default="integrated_output",
171         help="Folder for CSV, DB, plots, and summary JSON",
172     )
173     parser.add_argument(
174         "--rows", "-r", type=int, default=60, help="Number of rows to generate"
175     )
176     parser.add_argument(
177         "--api-days", type=int, default=5, help="Mock API days to generate"
178     )
179     parser.add_argument(
180         "--jobs", type=int, default=6, help="Parallel jobs to run"
181     )
182     return parser.parse_args()

```

```
183
184 def main():
185     args = parse_args()
186     output_dir = Path(args.output)
187     data_dir = output_dir / "data"
188     plots_dir = output_dir / "plots"
189     output_dir.mkdir(parents=True, exist_ok=True)
190
191     print("=== SIMPLIFIED INTEGRATION DEMO ===")
192     print(f"Output directory: {output_dir.resolve()}")
193
194     # 1) Dataset -> CSV + SQLite
195     df = create_dataset(rows=args.rows)
196     csv_path = data_dir / "engineering_test_data.csv"
197     db_path = data_dir / "research_data.db"
198     save_csv(df, csv_path)
199     save_sqlite(df, db_path)
200
201     # 2) Mock API
202     api_df = mock_api_data(days=args.api_days)
203
204     # 3) Parallel jobs
205     _, elapsed = run_parallel_jobs(num_jobs=args.jobs)
206
207     # 4) Plots
208     make_plots(df, api_df, plots_dir)
209
210     # 5) Summary
211     summary = build_summary(
212         csv_path=csv_path,
213         db_path=db_path,
214         api_records=len(api_df),
215         parallel_time=elapsed,
216         output_dir=output_dir,
217     )
218     summary_path = output_dir / "integrated_system_report.json"
219     with summary_path.open("w", encoding="utf-8") as f:
220         json.dump(summary, f, indent=2)
221     print(f"[SUMMARY] Saved {summary_path}")
222     print("Done.")
223
224
225 if __name__ == "__main__":
226     main()
```

Listing 5: Integration demo: DB + APIs + parallel

