**Phase 3: Data Pipeline & Automation - 90 Minutes**

## **🏗️ Understanding Production-Ready Scraping**

**What Makes a Scraper "Professional"?**

Moving from hobby scripts to client-ready systems requires:

* **Reliability**: Handles failures gracefully
* **Scalability**: Processes multiple sites/products simultaneously
* **Data Quality**: Validates and cleans extracted data
* **Monitoring**: Tracks performance and errors
* **Automation**: Runs without human intervention

### The Data Pipeline Concept

Think of your scraper as a factory assembly line:

1. **Input**: URLs to scrape
2. **Processing**: Extract and clean data
3. **Validation**: Ensure data quality
4. **Storage**: Save to database/files
5. **Output**: Deliver to client/API

## 📊 Minutes 0-30: Structured Data Extraction & Validation

**The Data Quality Problem**

Raw scraped data is messy:

* Extra whitespace and formatting
* Missing values and inconsistent formats
* Currency symbols and units mixed with numbers
* Duplicate or malformed entries

### Building a Data Processor

import re

import json

from datetime import datetime

from typing import Dict, List, Optional

from dataclasses import dataclass

@dataclass

class ProductData:

"""Structured product data with validation"""

title: str

price: Optional[float]

currency: str

rating: Optional[float]

review\_count: Optional[int]

availability: str

image\_url: Optional[str]

product\_url: str

scraped\_at: datetime

class DataProcessor:

"""Handles data cleaning and validation"""

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

# Price regex patterns for different currencies

self.price\_patterns = {

'USD': r'[\$]?([0-9,]+\.?[0-9]\*)',

'EUR': r'[€]?([0-9,]+\.?[0-9]\*)',

'GBP': r'[£]?([0-9,]+\.?[0-9]\*)',

'INR': r'[₹]?([0-9,]+\.?[0-9]\*)'

}

self.rating\_pattern = r'([0-5]\.?[0-9]?)'

self.review\_pattern = r'([0-9,]+)'

def clean\_text(self, text: str) -> str:

"""Clean and normalize text data"""

if not text:

return ""

# Remove extra whitespace

text = re.sub(r'\s+', ' ', text.strip())

# Remove special characters but keep basic punctuation

text = re.sub(r'[^\w\s.,!?()-]', '', text)

return text

def extract\_price(self, price\_str: str) -> tuple[Optional[float], str]:

"""Extract price value and currency"""

if not price\_str:

return None, "USD"

# Clean the price string

price\_str = self.clean\_text(price\_str)

# Try to detect currency and extract price

for currency, pattern in self.price\_patterns.items():

match = re.search(pattern, price\_str)

if match:

# Remove commas and convert to float

price\_value = float(match.group(1).replace(',', ''))

return price\_value, currency

return None, "USD"

def extract\_rating(self, rating\_str: str) -> Optional[float]:

"""Extract rating value"""

if not rating\_str:

return None

match = re.search(self.rating\_pattern, rating\_str)

if match:

rating = float(match.group(1))

return rating if 0 <= rating <= 5 else None

return None

def extract\_review\_count(self, review\_str: str) -> Optional[int]:

"""Extract review count"""

if not review\_str:

return None

# Remove parentheses and extra text

review\_str = re.sub(r'[()]', '', review\_str)

match = re.search(self.review\_pattern, review\_str)

if match:

return int(match.group(1).replace(',', ''))

return None

def process\_product\_data(self, raw\_data: Dict) -> ProductData:

"""Convert raw scraped data to structured format"""

price, currency = self.extract\_price(raw\_data.get('price', ''))

rating = self.extract\_rating(raw\_data.get('rating', ''))

review\_count = self.extract\_review\_count(raw\_data.get('reviews', ''))

return ProductData(

title=self.clean\_text(raw\_data.get('title', '')),

price=price,

currency=currency,

rating=rating,

review\_count=review\_count,

availability=self.clean\_text(raw\_data.get('availability', '')),

image\_url=raw\_data.get('image\_url'),

product\_url=raw\_data.get('url', ''),

scraped\_at=datetime.now()

)

def validate\_product\_data(self, product: ProductData) -> List[str]:

"""Validate product data and return list of issues"""

issues = []

if not product.title:

issues.append("Missing product title")

if product.price is None:

issues.append("Missing or invalid price")

elif product.price <= 0:

issues.append("Price must be positive")

if product.rating is not None and (product.rating < 0 or product.rating > 5):

issues.append("Rating must be between 0 and 5")

if not product.product\_url:

issues.append("Missing product URL")

return issues

**Using the Data Processor**

async def scrape\_with\_data\_processing(page, url):

"""Scrape with proper data processing"""

processor = DataProcessor()

try:

# Extract raw data

raw\_data = {

'title': await page.text\_content('h1'),

'price': await page.text\_content('.price, .a-price-whole'),

'rating': await page.text\_content('.rating, .a-icon-alt'),

'reviews': await page.text\_content('.review-count'),

'availability': await page.text\_content('.availability'),

'image\_url': await page.get\_attribute('img.main-image', 'src'),

'url': url

}

# Process and validate

product = processor.process\_product\_data(raw\_data)

issues = processor.validate\_product\_data(product)

if issues:

print(f"Data quality issues: {issues}")

# Decide whether to save or discard based on issues

return product

except Exception as e:

print(f"Error processing data: {e}")

return None

**Why This Matters:**

* Clean data is easier to analyze
* Validation catches errors early
* Structured data integrates better with databases
* Clients trust consistent, validated data

**🔄 Minutes 30-60: Error Recovery & Retry Strategies**

**The Reality of Web Scraping**

Things go wrong constantly:

* Network timeouts
* Pages that don't load
* Rate limiting
* Server errors
* Site structure changes

**Building a Robust Retry System**

import asyncio

import random

from enum import Enum

from typing import Callable, Any

from dataclasses import dataclass

class RetryStrategy(Enum):

EXPONENTIAL\_BACKOFF = "exponential"

FIXED\_DELAY = "fixed"

RANDOM\_DELAY = "random"

@dataclass

class RetryConfig:

max\_attempts: int = 3

strategy: RetryStrategy = RetryStrategy.EXPONENTIAL\_BACKOFF

base\_delay: float = 1.0

max\_delay: float = 60.0

backoff\_factor: float = 2.0

class RetryHandler:

"""Handles retry logic for failed scraping attempts"""

def \_\_init\_\_(self, config: RetryConfig):

self.config = config

async def execute\_with\_retry(self, func: Callable, \*args, \*\*kwargs) -> Any:

"""Execute function with retry logic"""

last\_exception = None

for attempt in range(self.config.max\_attempts):

try:

result = await func(\*args, \*\*kwargs)

if result is not None: # Success

return result

except Exception as e:

last\_exception = e

print(f"Attempt {attempt + 1} failed: {e}")

if attempt < self.config.max\_attempts - 1:

delay = self.\_calculate\_delay(attempt)

print(f"Retrying in {delay:.2f} seconds...")

await asyncio.sleep(delay)

# All attempts failed

raise Exception(f"All {self.config.max\_attempts} attempts failed. Last error: {last\_exception}")

def \_calculate\_delay(self, attempt: int) -> float:

"""Calculate delay based on retry strategy"""

if self.config.strategy == RetryStrategy.EXPONENTIAL\_BACKOFF:

delay = self.config.base\_delay \* (self.config.backoff\_factor \*\* attempt)

return min(delay, self.config.max\_delay)

elif self.config.strategy == RetryStrategy.FIXED\_DELAY:

return self.config.base\_delay

elif self.config.strategy == RetryStrategy.RANDOM\_DELAY:

return random.uniform(self.config.base\_delay, self.config.max\_delay)

return self.config.base\_delay

**Building a Robust Retry System**

import asyncio

import random

from enum import Enum

from typing import Callable, Any

from dataclasses import dataclass

class RetryStrategy(Enum):

EXPONENTIAL\_BACKOFF = "exponential"

FIXED\_DELAY = "fixed"

RANDOM\_DELAY = "random"

@dataclass

class RetryConfig:

max\_attempts: int = 3

strategy: RetryStrategy = RetryStrategy.EXPONENTIAL\_BACKOFF

base\_delay: float = 1.0

max\_delay: float = 60.0

backoff\_factor: float = 2.0

class RetryHandler:

"""Handles retry logic for failed scraping attempts"""

def \_\_init\_\_(self, config: RetryConfig):

self.config = config

async def execute\_with\_retry(self, func: Callable, \*args, \*\*kwargs) -> Any:

"""Execute function with retry logic"""

last\_exception = None

for attempt in range(self.config.max\_attempts):

try:

result = await func(\*args, \*\*kwargs)

if result is not None: # Success

return result

except Exception as e:

last\_exception = e

print(f"Attempt {attempt + 1} failed: {e}")

if attempt < self.config.max\_attempts - 1:

delay = self.\_calculate\_delay(attempt)

print(f"Retrying in {delay:.2f} seconds...")

await asyncio.sleep(delay)

# All attempts failed

raise Exception(f"All {self.config.max\_attempts} attempts failed. Last error: {last\_exception}")

def \_calculate\_delay(self, attempt: int) -> float:

"""Calculate delay based on retry strategy"""

if self.config.strategy == RetryStrategy.EXPONENTIAL\_BACKOFF:

delay = self.config.base\_delay \* (self.config.backoff\_factor \*\* attempt)

return min(delay, self.config.max\_delay)

elif self.config.strategy == RetryStrategy.FIXED\_DELAY:

return self.config.base\_delay

elif self.config.strategy == RetryStrategy.RANDOM\_DELAY:

return random.uniform(self.config.base\_delay, self.config.max\_delay)

return self.config.base\_delay

**Explanation of the Python Retry Mechanism**

This code defines a system for retrying asynchronous function calls that might fail due to temporary issues. It allows you to configure different strategies for how long to wait between retries.

**1. RetryStrategy Enum**

* **Purpose:** This is an Enum (enumeration) that defines the different types of delay strategies you can use between retry attempts.
* **Values:**
  + EXPONENTIAL\_BACKOFF: The delay between attempts increases exponentially. This is a common strategy to avoid overwhelming a service and to give it more time to recover.
  + FIXED\_DELAY: The delay between attempts remains constant.
  + RANDOM\_DELAY: The delay between attempts is a random value within a specified range. This can help distribute requests and avoid "thundering herd" problems if many clients are retrying simultaneously.

**2. RetryConfig Dataclass**

* **Purpose:** This dataclass holds all the configuration parameters for the retry logic. It provides default values for convenience.
* **Attributes:**
  + max\_attempts: The maximum number of times the function will be attempted, including the initial call. (Default: 3)
  + strategy: The RetryStrategy to use for calculating delays. (Default: EXPONENTIAL\_BACKOFF)
  + base\_delay: The initial delay in seconds for EXPONENTIAL\_BACKOFF and RANDOM\_DELAY, and the fixed delay for FIXED\_DELAY. (Default: 1.0 second)
  + max\_delay: The maximum delay in seconds allowed for EXPONENTIAL\_BACKOFF and the upper bound for RANDOM\_DELAY. This prevents delays from growing indefinitely large. (Default: 60.0 seconds)
  + backoff\_factor: The factor by which the delay increases in EXPONENTIAL\_BACKOFF. For example, a factor of 2.0 means the delay doubles with each attempt. (Default: 2.0)

**3. RetryHandler Class**

* **Purpose:** This is the main class that encapsulates the retry logic.
* **\_\_init\_\_(self, config: RetryConfig):**
  + **Purpose:** The constructor initializes the RetryHandler instance with a RetryConfig object. This config dictates how retries will behave for this handler.
* **async execute\_with\_retry(self, func: Callable, \*args, \*\*kwargs) -> Any:**
  + **Purpose:** This is the core asynchronous method that executes a given function (func) with the configured retry logic.
  + **Parameters:**
    - func: The asynchronous function (or any callable) that you want to execute and potentially retry.
    - \*args, \*\*kwargs: Arbitrary positional and keyword arguments to pass to func.
  + **How it works:**
    - **Initialization:** It keeps track of the last\_exception encountered.
    - **Attempt Loop:** It iterates for self.config.max\_attempts times.
    - **try-except Block:**
      * Inside the try block, it awaits the execution of func with the provided arguments.
      * **Success Condition:** If func returns a non-None result, it means the operation was successful, and the result is immediately returned.
      * **Failure Handling:** If an Exception occurs (or func returns None and is considered a failure), the exception is caught, stored as last\_exception, and a message is printed.
    - **Delay before Retry:** If it's not the last attempt, it calculates the delay using \_calculate\_delay and then await asyncio.sleep(delay) to pause execution for that duration.
    - **Final Failure:** If the loop completes (meaning all attempts failed), it raises a new Exception indicating that all attempts were unsuccessful, including the last\_exception for debugging.
* **\_calculate\_delay(self, attempt: int) -> float:**
  + **Purpose:** This private helper method calculates the delay duration based on the chosen retry\_strategy and the current attempt number.
  + **Logic:**
    - **EXPONENTIAL\_BACKOFF:** Calculates base\_delay \* (backoff\_factor \*\* attempt). It then ensures this calculated delay does not exceed max\_delay.
    - **FIXED\_DELAY:** Simply returns base\_delay.
    - **RANDOM\_DELAY:** Returns a random float between base\_delay and max\_delay.
    - **Default:** If an unknown strategy is somehow encountered, it defaults to base\_delay.

**How to Use It**

You would typically instantiate RetryConfig (customizing if needed), then create a RetryHandler with that config, and finally call execute\_with\_retry with the asynchronous function you want to make resilient:

# Example Usage (not part of the provided code, but shows how to use it)

import asyncio

async def unreliable\_function(succeed\_on\_attempt: int):

"""A simulated function that might fail a few times."""

global call\_count

call\_count += 1

print(f" Unreliable function called (attempt {call\_count})")

if call\_count < succeed\_on\_attempt:

print(f" Simulating failure on attempt {call\_count}...")

raise ValueError("Simulated network error")

print(f" Simulating success on attempt {call\_count}!")

return "Data fetched successfully!"

async def main():

global call\_count

call\_count = 0

# Example 1: Exponential Backoff

print("\n--- Testing Exponential Backoff ---")

config\_exp = RetryConfig(max\_attempts=4, strategy=RetryStrategy.EXPONENTIAL\_BACKOFF, base\_delay=0.5, backoff\_factor=2)

handler\_exp = RetryHandler(config\_exp)

try:

result = await handler\_exp.execute\_with\_retry(unreliable\_function, succeed\_on\_attempt=3)

print(f"Result (Exponential): {result}")

except Exception as e:

print(f"Failed (Exponential): {e}")

call\_count = 0

# Example 2: Fixed Delay

print("\n--- Testing Fixed Delay ---")

config\_fixed = RetryConfig(max\_attempts=3, strategy=RetryStrategy.FIXED\_DELAY, base\_delay=1.0)

handler\_fixed = RetryHandler(config\_fixed)

try:

result = await handler\_fixed.execute\_with\_retry(unreliable\_function, succeed\_on\_attempt=2)

print(f"Result (Fixed): {result}")

except Exception as e:

print(f"Failed (Fixed): {e}")

call\_count = 0

# Example 3: All attempts fail

print("\n--- Testing All Attempts Fail ---")

config\_fail = RetryConfig(max\_attempts=2, strategy=RetryStrategy.FIXED\_DELAY, base\_delay=0.1)

handler\_fail = RetryHandler(config\_fail)

try:

result = await handler\_fail.execute\_with\_retry(unreliable\_function, succeed\_on\_attempt=5) # Will always fail within 2 attempts

print(f"Result (Fail): {result}")

except Exception as e:

print(f"Failed (Fail): {e}")

# If running directly:

# if \_\_name\_\_ == "\_\_main\_\_":

# asyncio.run(main())

**Benefits**

* **Resilience:** Makes your asynchronous operations more robust against transient failures.
* **Configurability:** Allows easy customization of retry behavior (number of attempts, delay strategy, delays).
* **Asynchronous Support:** Designed to work seamlessly with asyncio for non-blocking operations.
* **Clean Code:** Centralizes retry logic, preventing repetition in your application code.

### Smart Error Classification

class ErrorClassifier:

"""Classifies errors to determine appropriate response"""

RETRYABLE\_ERRORS = [

'TimeoutError',

'NetworkError',

'ConnectionError',

'HTTPError'

]

RATE\_LIMIT\_INDICATORS = [

'rate limit',

'too many requests',

'429',

'blocked',

'captcha'

]

PERMANENT\_ERRORS = [

'404',

'not found',

'access denied',

'forbidden'

]

@classmethod

def should\_retry(cls, error: Exception) -> bool:

"""Determine if error is worth retrying"""

error\_str = str(error).lower()

# Don't retry permanent errors

if any(perm in error\_str for perm in cls.PERMANENT\_ERRORS):

return False

# Retry network-related errors

if any(retry in error\_str for retry in cls.RETRYABLE\_ERRORS):

return True

# Handle rate limiting with longer delays

if any(rate in error\_str for rate in cls.RATE\_LIMIT\_INDICATORS):

return True

return False

@classmethod

def get\_retry\_delay(cls, error: Exception) -> float:

"""Get appropriate retry delay based on error type"""

error\_str = str(error).lower()

# Longer delays for rate limiting

if any(rate in error\_str for rate in cls.RATE\_LIMIT\_INDICATORS):

return random.uniform(60, 300) # 1-5 minutes

# Standard delays for other errors

return random.uniform(5, 15)

**Complete Error Recovery Example**

async def robust\_scrape\_product(url: str, max\_retries: int = 3):

"""Scrape product with comprehensive error handling"""

retry\_config = RetryConfig(

max\_attempts=max\_retries,

strategy=RetryStrategy.EXPONENTIAL\_BACKOFF,

base\_delay=2.0,

max\_delay=60.0

)

retry\_handler = RetryHandler(retry\_config)

async def \_scrape\_attempt():

async with async\_playwright() as p:

browser, context = await create\_stealth\_browser(p)

page = await context.new\_page()

try:

# Navigate with timeout

await page.goto(url, wait\_until='networkidle', timeout=30000)

# Wait for key elements

await page.wait\_for\_selector('h1', timeout=15000)

# Extract data

product\_data = await scrape\_with\_data\_processing(page, url)

if not product\_data:

raise Exception("No data extracted")

return product\_data

except Exception as e:

if not ErrorClassifier.should\_retry(e):

raise Exception(f"Permanent error: {e}")

delay = ErrorClassifier.get\_retry\_delay(e)

await asyncio.sleep(delay)

raise e

finally:

await browser.close()

try:

return await retry\_handler.execute\_with\_retry(\_scrape\_attempt)

except Exception as e:

print(f"Failed to scrape {url} after {max\_retries} attempts: {e}")

return None

**Why This Approach Works:**

* **Intelligent retries**: Only retries when it makes sense
* **Exponential backoff**: Reduces server load
* **Error classification**: Different strategies for different problems
* **Graceful degradation**: Continues working even with some failures

## 🗄️ Minutes 60-90: Database Integration & Scheduling

### Professional Data Storage

Moving beyond JSON files to proper database storage:

import sqlite3

import json

from datetime import datetime

from typing import List, Optional

class ProductDatabase:

"""Handles product data storage and retrieval"""

def \_\_init\_\_(self, db\_path: str = "products.db"):

self.db\_path = db\_path

self.init\_database()

def init\_database(self):

"""Initialize database tables"""

conn = sqlite3.connect(self.db\_path)

cursor = conn.cursor()

cursor.execute("""

CREATE TABLE IF NOT EXISTS products (

id INTEGER PRIMARY KEY AUTOINCREMENT,

title TEXT NOT NULL,

price REAL,

currency TEXT,

rating REAL,

review\_count INTEGER,

availability TEXT,

image\_url TEXT,

product\_url TEXT UNIQUE,

scraped\_at TIMESTAMP,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

""")

cursor.execute("""

CREATE TABLE IF NOT EXISTS scraping\_logs (

id INTEGER PRIMARY KEY AUTOINCREMENT,

url TEXT,

status TEXT,

error\_message TEXT,

duration REAL,

scraped\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

""")

conn.commit()

conn.close()

def save\_product(self, product: ProductData) -> bool:

"""Save product to database"""

conn = sqlite3.connect(self.db\_path)

cursor = conn.cursor()

try:

cursor.execute("""

INSERT OR REPLACE INTO products

(title, price, currency, rating, review\_count, availability,

image\_url, product\_url, scraped\_at)

VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)

""", (

product.title,

product.price,

product.currency,

product.rating,

product.review\_count,

product.availability,

product.image\_url,

product.product\_url,

product.scraped\_at

))

conn.commit()

return True

except Exception as e:

print(f"Error saving product: {e}")

return False

finally:

conn.close()

def log\_scraping\_attempt(self, url: str, status: str, error: str = None, duration: float = None):

"""Log scraping attempt for monitoring"""

conn = sqlite3.connect(self.db\_path)

cursor = conn.cursor()

cursor.execute("""

INSERT INTO scraping\_logs (url, status, error\_message, duration)

VALUES (?, ?, ?, ?)

""", (url, status, error, duration))

conn.commit()

conn.close()

def get\_products\_by\_price\_range(self, min\_price: float, max\_price: float) -> List[Dict]:

"""Query products by price range"""

conn = sqlite3.connect(self.db\_path)

cursor = conn.cursor()

cursor.execute("""

SELECT \* FROM products

WHERE price BETWEEN ? AND ?

ORDER BY price ASC

""", (min\_price, max\_price))

rows = cursor.fetchall()

conn.close()

# Convert to dictionaries

columns = [desc[0] for desc in cursor.description]

return [dict(zip(columns, row)) for row in rows]

def get\_scraping\_stats(self) -> Dict:

"""Get scraping performance statistics"""

conn = sqlite3.connect(self.db\_path)

cursor = conn.cursor()

cursor.execute("""

SELECT

status,

COUNT(\*) as count,

AVG(duration) as avg\_duration

FROM scraping\_logs

GROUP BY status

""")

stats = {}

for row in cursor.fetchall():

stats[row[0]] = {'count': row[1], 'avg\_duration': row[2]}

conn.close()

return stats

**Automated Scheduling System**

import schedule

import time

from datetime import datetime, timedelta

from typing import List

import asyncio

class ScrapingScheduler:

"""Handles automated scraping schedules"""

def \_\_init\_\_(self, db: ProductDatabase):

self.db = db

self.is\_running = False

def schedule\_product\_updates(self, urls: List[str], interval\_hours: int = 24):

"""Schedule regular product updates"""

async def scrape\_products():

print(f"Starting scheduled scraping of {len(urls)} products...")

for url in urls:

try:

start\_time = time.time()

product = await robust\_scrape\_product(url)

duration = time.time() - start\_time

if product:

self.db.save\_product(product)

self.db.log\_scraping\_attempt(url, "success", duration=duration)

print(f"✅ Successfully scraped: {product.title}")

else:

self.db.log\_scraping\_attempt(url, "failed", "No data extracted")

print(f"❌ Failed to scrape: {url}")

# Delay between products

await asyncio.sleep(random.uniform(5, 15))

except Exception as e:

self.db.log\_scraping\_attempt(url, "error", str(e))

print(f"❌ Error scraping {url}: {e}")

# Schedule the scraping

schedule.every(interval\_hours).hours.do(lambda: asyncio.run(scrape\_products()))

print(f"Scheduled scraping every {interval\_hours} hours")

def schedule\_price\_monitoring(self, urls: List[str], price\_threshold: float):

"""Schedule price monitoring with alerts"""

async def monitor\_prices():

print("Checking prices for significant changes...")

for url in urls:

try:

# Get current price

product = await robust\_scrape\_product(url)

if not product or product.price is None:

continue

# Get previous price from database

conn = sqlite3.connect(self.db.db\_path)

cursor = conn.cursor()

cursor.execute("""

SELECT price FROM products

WHERE product\_url = ?

ORDER BY scraped\_at DESC

LIMIT 1 OFFSET 1

""", (url,))

previous\_price = cursor.fetchone()

conn.close()

if previous\_price:

old\_price = previous\_price[0]

price\_change = abs(product.price - old\_price) / old\_price

if price\_change > price\_threshold:

print(f"🚨 Price alert: {product.title}")

print(f" Old price: ${old\_price:.2f}")

print(f" New price: ${product.price:.2f}")

print(f" Change: {price\_change:.1%}")

# Save current price

self.db.save\_product(product)

except Exception as e:

print(f"Error monitoring {url}: {e}")

# Schedule price monitoring

schedule.every(6).hours.do(lambda: asyncio.run(monitor\_prices()))

def run\_scheduler(self):

"""Run the scheduler"""

self.is\_running = True

print("Starting scraping scheduler...")

while self.is\_running:

schedule.run\_pending()

time.sleep(60) # Check every minute

def stop\_scheduler(self):

"""Stop the scheduler"""

self.is\_running = False

print("Stopping scheduler...")

**Refactored Approach Example**

Here's an example of how the scheduler could be rewritten to be fully asynchronous, addressing the issues above. This version is more efficient, concurrent, and better structured.

import asyncio

import random

from typing import List

# Assume ProductDatabase and robust\_scrape\_product are defined elsewhere

class AsyncScrapingScheduler:

"""Handles automated scraping schedules using a native asyncio loop."""

def \_\_init\_\_(self, db: ProductDatabase):

self.db = db

self.\_tasks = [] # To hold running asyncio tasks

async def \_product\_update\_worker(self, urls: List[str], interval\_hours: int):

"""Asynchronous worker for scraping products."""

print(f"✅ Product update worker started. Interval: {interval\_hours} hours.")

while True:

print(f"Starting scheduled scraping of {len(urls)} products...")

# Create a list of scraping tasks to run concurrently

scrape\_tasks = [robust\_scrape\_product(url) for url in urls]

# Run all scraping tasks concurrently and get results

results = await asyncio.gather(\*scrape\_tasks, return\_exceptions=True)

for i, result in enumerate(results):

url = urls[i]

if isinstance(result, Exception):

# Handle exceptions returned by gather

self.db.log\_scraping\_attempt(url, "error", str(result))

print(f"❌ Error scraping {url}: {result}")

elif result:

# Process successful scrapes

self.db.save\_product(result)

self.db.log\_scraping\_attempt(url, "success")

print(f"✅ Successfully scraped: {result.title}")

else:

self.db.log\_scraping\_attempt(url, "failed", "No data extracted")

print(f"❌ Failed to scrape: {url}")

print("Scheduled scraping finished. Waiting for next run...")

await asyncio.sleep(interval\_hours \* 3600)

# A similar async worker for price monitoring would be created here

# e.g., async def \_price\_monitoring\_worker(...)

def schedule\_product\_updates(self, urls: List[str], interval\_hours: int = 24):

"""Schedules the product update worker to run."""

# Create a task for the worker and add it to our list

task = asyncio.create\_task(

self.\_product\_update\_worker(urls, interval\_hours)

)

self.\_tasks.append(task)

print(f"Scheduled product updates every {interval\_hours} hours.")

async def run(self):

"""Run all scheduled tasks until they are cancelled."""

print("Starting async scheduler...")

if not self.\_tasks:

print("No tasks scheduled. Exiting.")

return

# This will run forever, or until the tasks are cancelled

await asyncio.gather(\*self.\_tasks)

def stop(self):

"""Stops all running scheduler tasks."""

print("Stopping scheduler...")

for task in self.\_tasks:

task.cancel()

self.\_tasks = []

print("Scheduler stopped.")

# Example Usage:

# async def main():

# db = ProductDatabase()

# scheduler = AsyncScrapingScheduler(db)

# urls = ["http://example.com/product1", "http://example.com/product2"]

# scheduler.schedule\_product\_updates(urls, interval\_hours=1)

# await scheduler.run()

#

# if \_\_name\_\_ == "\_\_main\_\_":

# try:

# asyncio.run(main())

# except KeyboardInterrupt:

# print("Program interrupted.")

**Complete Production System**

**import asyncio**

**import json**

**from datetime import datetime**

**from typing import List**

**class ProductScrapingPipeline:**

**"""Complete production scraping pipeline"""**

**def \_\_init\_\_(self, db\_path: str = "products.db"):**

**self.db = ProductDatabase(db\_path)**

**self.scheduler = ScrapingScheduler(self.db)**

**self.processor = DataProcessor()**

**async def batch\_scrape(self, urls: List[str], max\_concurrent: int = 5):**

**"""Scrape multiple URLs concurrently"""**

**# Limit concurrent requests**

**semaphore = asyncio.Semaphore(max\_concurrent)**

**async def scrape\_with\_semaphore(url):**

**async with semaphore:**

**return await robust\_scrape\_product(url)**

**# Create tasks for all URLs**

**tasks = [scrape\_with\_semaphore(url) for url in urls]**

**# Execute with progress tracking**

**results = []**

**for i, task in enumerate(asyncio.as\_completed(tasks)):**

**result = await task**

**results.append(result)**

**print(f"Progress: {i+1}/{len(urls)} completed")**

**return results**

**async def export\_data(self, format: str = "json", filename: str = None):**

**"""Export scraped data in various formats"""**

**if filename is None:**

**filename = f"products\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.{format}"**

**# Get all products from database**

**conn = sqlite3.connect(self.db.db\_path)**

**cursor = conn.cursor()**

**cursor.execute("SELECT \* FROM products ORDER BY scraped\_at DESC")**

**products = []**

**columns = [desc[0] for desc in cursor.description]**

**for row in cursor.fetchall():**

**products.append(dict(zip(columns, row)))**

**conn.close()**

**if format == "json":**

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

**json.dump(products, f, indent=2, default=str)**

**elif format == "csv":**

**import csv**

**with open(filename, 'w', newline='') as f:**

**if products:**

**writer = csv.DictWriter(f, fieldnames=products[0].keys())**

**writer.writeheader()**

**writer.writerows(products)**

**print(f"Exported {len(products)} products to {filename}")**

**def generate\_report(self):**

**"""Generate scraping performance report"""**

**stats = self.db.get\_scraping\_stats()**

**print("\n📊 Scraping Performance Report")**

**print("=" \* 40)**

**total\_attempts = sum(stat['count'] for stat in stats.values())**

**success\_rate = (stats.get('success', {}).get('count', 0) / total\_attempts \* 100) if total\_attempts > 0 else 0**

**print(f"Total scraping attempts: {total\_attempts}")**

**print(f"Success rate: {success\_rate:.1f}%")**

**print(f"Average duration: {stats.get('success', {}).get('avg\_duration', 0):.2f}s")**

**for status, data in stats.items():**

**print(f"{status.capitalize()}: {data['count']} attempts")**

**# Usage Example**

**async def main():**

**pipeline = ProductScrapingPipeline()**

**# URLs to scrape**

**urls = [**

**"https://example-store.com/product/1",**

**"https://example-store.com/product/2",**

**"https://example-store.com/product/3"**

**]**

**# Batch scrape**

**results = await pipeline.batch\_scrape(urls, max\_concurrent=3)**

**# Save to database**

**for result in results:**

**if result:**

**pipeline.db.save\_product(result)**

**# Export data**

**await pipeline.export\_data("json")**

**await pipeline.export\_data("csv")**

**# Generate report**

**pipeline.generate\_report()**

**# Schedule regular updates**

**pipeline.scheduler.schedule\_product\_updates(urls, interval\_hours=24)**

**pipeline.scheduler.schedule\_price\_monitoring(urls, price\_threshold=0.1)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**asyncio.run(main())**

**🎯 Key Benefits of This Pipeline**

**1. Reliability**

* Automatic retry with intelligent backoff
* Error classification and handling
* Comprehensive logging

**2. Scalability**

* Concurrent processing with rate limiting
* Database storage for large datasets
* Batch processing capabilities

**3. Maintainability**

* Modular design with clear separation
* Comprehensive error handling
* Performance monitoring

**4. Production-Ready**

* Automated scheduling
* Data export in multiple formats
* Performance reporting

**✅ Phase 3 Mastery Checklist**

You've mastered the data pipeline when you can:

* Process and validate scraped data automatically
* Handle errors gracefully with intelligent retries
* Store data in a proper database
* Schedule automated scraping runs
* Monitor performance and generate reports
* Export data in multiple formats
* Handle concurrent scraping safely

**Success Indicator:** You can deploy a scraping system that runs automatically for weeks without manual intervention.

**🎯 Ready for Phase 4?**

You're ready for **Phase 4: Scaling & Professional Features** when you can:

* Build complete data pipelines
* Handle errors and retries professionally
* Use databases for data storage
* Schedule automated runs
* Monitor and report on performance

**Next Up:** We'll scale your system to handle enterprise-level requirements with multi-threading, API development, and client delivery!

*💡 Pro Tip: Phase 3 is where you transition from "scripter" to "engineer." Focus on building systems that work reliably without your constant attention.*