

Assignment 1: PricePulse – E-Commerce Price Tracker & Smart Comparator

Objective

Build a full-stack web application that allows users to:

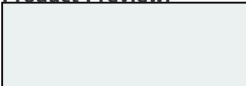
1. Enter an Amazon product URL.
2. Automatically track and record the product's price every 30 minutes or 1 hour.
3. Visualize the price trend on a graph over time.

PricePulse - E-Commerce Price Tracker

Enter Amazon Product URL:

Track

Product Preview:



Samsung Galaxy M14
Current Price: ₹13,499

Price History Graph:

[Graph Placeholder]

Available on Other Platforms (Bonus):

- Flipkart: ₹13,299
- Meesho: ₹13,499
- BigBasket: Not Available

Bonus: Use generative AI to identify and compare the same product across other platforms like Flipkart, Meesho, BigBasket, or Swiggy Instamart.

Part 1: Core Assignment – Amazon Price Tracker

Features to Implement

- A live web application with:
 - **Input field** for Amazon product URL.
 - **Submit button** to start tracking.
 - **Live graph** showing historical prices over time (line or bar chart).
 - **Display** of current price and link preview (name/image).

Backend Requirements

- Schedule a scraping job every hour or 30 minutes (e.g., using cron, Celery Beat, or APScheduler).
 - Store results in a local database (SQLite, PostgreSQL, or Firebase).
 - Use a scraper tool or library (e.g., **BeautifulSoup**, **Playwright**, **Selenium**, **Scrapy**) or **API if available**.
 - Show error handling (e.g., product unavailable, request blocked).
-

Part 2: Bonus Assignment – Multi-Platform & Generative AI

Objective

- Use LLMs or APIs (e.g., ChatGPT, Gemini, or Serper.dev) to:
 - **Extract product metadata** (name, brand, model) from the given Amazon URL.
 - **Search for the same product on other platforms** using that metadata (via scraping or Google Search APIs).
 - **Scrape their prices** and list comparisons in a table.
 - **Optional:** Add a price history graph for other platforms too.
-

Tools Allowed

- Backend: Python (FastAPI / Flask / Django), Node.js, or others.
 - Frontend: React / HTML+JS / any modern web framework.
 - Scheduler: Cron, Celery, or cloud functions.
 - Database: SQLite, PostgreSQL, Firebase, Supabase, etc.
 - Deployment: Render, Railway, Vercel, Netlify, or own VPS.
 - AI (Bonus): OpenAI, Gemini, or other LLM-based API for searching.
-

Deliverables

1. **Live Web App URL** (hosted)
 2. **GitHub Repo** with:
 - Source code
 - **README.md** with setup instructions
 3. **Video Demo** (3–5 min) explaining:
 - How the app works
 - Technology choices
 - Challenges faced
 4. **Flow Diagram / Architecture Doc**
 - Show backend jobs, scraping flow, and UI logic
-

Evaluation Criteria

Criteria	Weightage
Core Functionality (Scraping, Graph)	30%
UI/UX & Responsiveness	20%
Code Structure & Documentation	20%
Scheduler & Automation Reliability	15%
Bonus (AI search + multi-platform)	15%

Sample Scenario

User enters:

<https://www.amazon.in/dp/B0CV7KZLL4/>

- App extracts product name: **"Samsung Galaxy M14"**
 - Tracks its price every 30 minutes from Amazon
 - Optionally find it on Flipkart, Meesho, BigBasket, etc.
-

Additional Bonus Feature: Price Drop Alert via Email

Objective

Allow users to **set a target price** for the product. If the product's price drops **below this target**, an **email alert** should automatically be sent to them.

Feature Scope

- Add an **optional input field** on the web page:
“Notify me when price drops below ₹_____”
- Store user email + price threshold in the backend.
- On each scheduled price check:
 - Compare current price with the stored threshold.
 - If the price is below the threshold **and not alerted yet**, trigger an email.
- Email should include:
 - Product name and image
 - Current price and threshold
 - Link to view the product

Tools/Tech Suggestions

- **Email Service:** Use services like:
 - [SendGrid](#)
 - [Mailgun](#)
 - SMTP via Gmail for basic setup
- **Backend Integration:**
 - Add an alerts table or schema in the database
 - Maintain a flag to prevent repeat alerts for the same drop

Bonus UI Additions

- A field: Target Price (next to URL input)
- A field: Your Email
- Status: “Alert scheduled” or “Alert sent!”

7-Day Task Breakdown: PricePulse Assignment

Day 1: Project Setup + Research

Objective: Lay the foundation, understand the stack, and decide tools.

- Research web scraping techniques for Amazon (BeautifulSoup, Playwright, Selenium).
- Choose the tech stack (e.g., Flask + SQLite + JS/Chart.js or React).
- Set up the GitHub repo and backend project structure.
- Create a basic HTML page with a form to submit an Amazon URL.

Deliverables:

- Tech stack selected
 - GitHub initialized
 - Skeleton frontend + backend structure
-

Day 2: Implement Amazon Scraper Script

Objective: Build and test the basic scraper.

- Implement a script to fetch product name and current price from an Amazon URL.
- Handle edge cases (redirects, blocked requests, unavailable pages).
- Run it locally for 1-2 product links.
- Store results in a local database (timestamp + price).

Deliverables:

- Working scraping script
 - Test product data stored in DB
 - Console/log evidence of scrape
-

Day 3: Scheduler + Backend Integration

Objective: Automate scraping and link to backend.

- Use APScheduler, Celery, or a cron job to run scrape every hour.
- Connect scraper to database write.
- Add basic backend routes (e.g., submit product URL, get data).

Deliverables:

- Fully automated hourly scraping
 - Database entries populating over time
 - REST endpoint for frontend to fetch historical data
-

Day 4: Frontend – Graph + Dashboard UI

Objective: Build frontend with input form and graph display.

- Create UI to:
 - Submit product URL
 - Show product preview
 - Display a chart (e.g., Chart.js or Google Charts) of price over time
- Make frontend dynamic (AJAX or fetch API)

Deliverables:

- Input form working
 - Graph showing price history
 - Live preview of scraped product
-

Day 5: Email Alert Feature

Objective: Implement target price email notifications.

- Add fields for user email + price threshold.
- Store alert data in DB.
- Compare price during each scrape and send email when threshold is hit.
- Use Gmail SMTP or SendGrid for alerts.

Deliverables:

- User can set alert
 - Email sent successfully (test scenario)
 - Alert system visible in logs/UI
-

Day 6: Bonus – Multi-platform Search via LLM (Optional)

Objective: Use AI to expand to other platforms (Flipkart, Meesho, etc.)

- Use OpenAI/Gemini API to extract product metadata (e.g., title, model).
- Use it to form search queries like: "Samsung Galaxy M14 site:flipkart.com"
- Scrape result links and prices if possible.
- Display alternate prices in table format.

Deliverables:

- Metadata extraction via LLM
 - Search results from at least 1 other platform
 - Optional price comparison table
-

Day 7: Polish + Submit

Objective: Final testing, bug fixing, and submission.

- Test across browsers and mobile (responsiveness).
- Add error messages, loading states, input validation.
- Record demo video (3–5 min).
- Write README .md, flow diagram, and submit the live URL.

Deliverables:

- Working, polished web app (live URL)
- GitHub repo with clean code & documentation
- Video walkthrough + flow diagram

Reach out at info@alfaleus.com if you have any questions.