QUICK SCHOLAR

NAME : *QUICK SCHOLAR*

IDENTIFIED PROBLEM : Students waste meaningful time when doing research for their assignments this time consumption is due to them looking for multiple sources in different disconnected platforms,websites etc

PROPOSED SOLUTION : Design a tool that pull data sources based on the input of the user from multiple sources/websites/data platforms and consolidate the information in one place,this will potentially save student time when looking for sources for their assignments

EXPECTED FEATURES :

1. **MVP(Minimum viable product)**
2. **UX / USER-INTERFACE**
3. **WEB HOSTING**
4. **WEB EXTENSION on(**Chrome , Bing and Firefox**)**

TIME EXPECTATION FOR EACH FEATURE :

FEATURE	**TIME EXPECTATION(*Approximately*)**
MVP(Minimum viable product)	**8–10 weeks (2–2.5 months)**
UX / USER-INTERFACE	**4–6 weeks (1–1.5 months)**
WEB HOSTING	**1–2 weeks**
WEB EXTENSION on(Chrome , Bing and Firefox**)**	**4–6 weeks (1–1.5 months)**
TOTAL ESTIMATED TIME	**5–6 months**

GOALS AND SUCCESS CRITERIA

SUCCESS

- The project shall be considered a success if it allows user to effortlessly search /type in input and receive a list of links ,summaries and data location
- The project shall be considered a success if all the initial expected features are fully and completely deployed.
- The project shall be considered a success it it has at least 10 users who are actively utilizing it and have positive feedback about the usability of the tool.
- The project shall be considered a success if it is fully documented on github and Linkedin
- The project shall be considered a success if the toll has no critical bugs in core flow.

GOALS

- Deploy all expected features within their expected timeframe
- Obtain at least 10 users who are proven to be using the tool during research on a regular basis.
- Have a functional and operation web extension that is integrated to the web app.

TARGET USER AND USER-PERSONA

- The main target user for the tool will be university students who frequently need citations for their academic work,secondary users might/will/can be high school students

  PRIMARY USER
  - University students

  SECONDARY USER
  - High school students

PAINPOINTS OF PRIMARY USER

- Constantly looking for sources on unrelated websites or platforms.
- Time consumed looking for sources that are not relevant to their assignment requirements

PAINPOINTS OF SECONDARY USERS

- Constantly looking for sources on unrelated websites or platforms.
- Time consumed looking for sources that are not relevant to their assignment requirements

CURRENT USER-WORKFLOW

- Open their assignments ,look for sources manually by searching both in google scholar and maybe youtube ,and manually read summaries of each source to check whether its relevant to their assignment.

DESIRED USER-WORKFLOW

- look for sources by simply clicking on the web-app link or using the web extension ,type the required input instantly and get 5 or 10 links of sources from both the internet and other academic data platforms with summaries.

CORE USER CASES AND USER FLOW

Core User Case 1: Search for Information

Use Case Name: Search for Research Sources

Actor: University student (primary), High school student (secondary)

Goal: Quickly find relevant sources for a research topic

Preconditions: User is on the QuickScholar homepage

Success Criteria: User receives a list of relevant links, summaries, and source info

User Flow:

1. User opens QuickScholar web app or extension
2. User enters topic keywords in the search box
3. User optionally selects sources/platforms to search (Google Scholar, Semantic Scholar, etc.)
4. User clicks “Search”
5. System displays a loading animation
6. Backend fetches data from selected sources

7. System parses, cleans, and ranks results

8. System displays results in a table/list with:

    - Title

    - Short summary

    - Link to original source

    - Date / author info

9. User scrolls results and selects relevant items

Core User Case 2: Filter / Refine Results

Use Case Name: Filter Results

Actor: Same

Goal: Narrow results to most relevant sources

Preconditions: Search results are displayed

Success Criteria: Filtered results show only items matching selected criteria

User Flow:

1. User clicks “Filter” or “Sort” options

2. User selects:
    - Date range
    - Source type
    - Relevance / keyword match
3. System updates results dynamically
4. User reviews the filtered list and selects items

Core User Case 3: Save / Export Results

Use Case Name: Save or Export Research Items

Actor: Same

Goal: Keep selected sources for later use

Preconditions: User has viewed search results

Success Criteria: User can save or export selected items successfully

User Flow:

1. User selects one or more results
2. User clicks “Save” or “Export”

3. System prompts for format:
    - CSV, PDF, JSON
    - Optionally: save to account if logged in
4. System confirms save/export success
5. User can access saved items later via dashboard

Core User Case 4: Browser Extension Search

Use Case Name: Search from Browser

Actor: Same

Goal: Perform QuickScholar search directly from any webpage

Preconditions: Extension is installed and active

Success Criteria: Query is sent to QuickScholar backend and results are displayed

User Flow:

1. User highlights text or clicks QuickScholar icon in browser toolbar
2. Extension captures highlighted text or allows manual query input
3. Extension sends query to backend

4. Backend fetches and processes results
5. Extension displays results in popup or redirects user to web app
6. User selects relevant items and optionally saves/exports

Core User Case 5: Account Management (Optional for MVP)

Use Case Name: User Accounts

Actor: Same

Goal: Access saved searches and results across sessions

Preconditions: User is logged out

Success Criteria: User can log in or create an account, and saved items persist

User Flow:

1. User clicks “Sign Up” or “Log In”
2. User provides email/password or OAuth login
3. Backend validates credentials
4. User is redirected to dashboard
5. User can view saved searches, export results, or continue new searches

SYSTEM ARCHITECTURE

1️⃣ What is System Architecture?

Definition (in simple terms):

System architecture is basically the **blueprint of your software**. It shows:

- **Components/parts of your system** (frontend, backend, database, APIs, browser extension, etc.)
- **How they interact** (who talks to whom, what data flows where)
- **Where logic lives** (backend vs frontend)

Think of it like building a house:

- You don't just start hammering nails.
- First you plan the rooms, doors, plumbing, wiring.
- System architecture is the *floor plan and wiring diagram* for your software.

Why it matters for your project:

- Shows the **big picture**, so you don’t get lost coding random features.
- Helps you **break the project into modules**, which you can build step by step.
- Helps you **avoid redesigning things halfway**, saving weeks of work.
- Makes it easier to **debug, scale, or add features** later.

2️⃣ How System Architecture Directs Project Creation

A well-thought architecture basically tells you:

- **Step 1:** Build the backend API first, because the frontend and extension depend on it.
- **Step 2:** Build the database structure, because both backend and frontend use it.
- **Step 3:** Build MVP features in the backend → then connect to frontend UI → then extension.
- **Step 4:** Test each module independently, then integrate.

Without architecture, you’re coding blindly. With it, you:

- Know which components to start with
- Know what to test first
- Know how data flows
- Know how to scale or add features later

3️⃣ QuickScholar System Architecture (Plan Version)

Here's how I'd structure it **at a high level** for your 6-month plan:

A. Components

1. **Frontend (Web App)**
   - Handles user interaction
   - Displays search box, results, filters
   - Optional: dashboards for saved/exported items
2. **Backend (Python, Flask/FastAPI)**

- Handles:
    - Search queries
    - Fetching data from external APIs (Google Scholar alternatives)
    - Parsing, cleaning, ranking results
    - Sending results back to frontend/extension
- Handles authentication if accounts are added

3. **Database**
    - Stores:
        - Saved searches / user selections
        - User accounts (optional)
        - Cached results (optional, for faster search)

    - Tech choice: SQLite for MVP → Postgres if scaling

4. **Browser Extension**
    - Captures user input from any webpage
    - Sends search query to backend
    - Displays results in popup or redirects to web app
5. **External Data Sources**
    - APIs or scrapers for:
        - Google Scholar alternatives
        - Semantic Scholar
        - CrossRef or other open educational sources

B. Data Flow

1. **User Interaction**
    - User enters query → frontend or extension sends query to backend API
2. **Backend Processing**
    - Backend receives query → queries external APIs or scrapers → parses data → ranks results
3. **Data Return**
    - Backend sends results to:
        - Web app frontend → displays to user
        - Browser extension → popup or redirect
4. **Optional Storage**
    - If user saves results:
        - Backend stores them in database

        - User can later retrieve/export them

C. Architectural Notes / Design Decisions

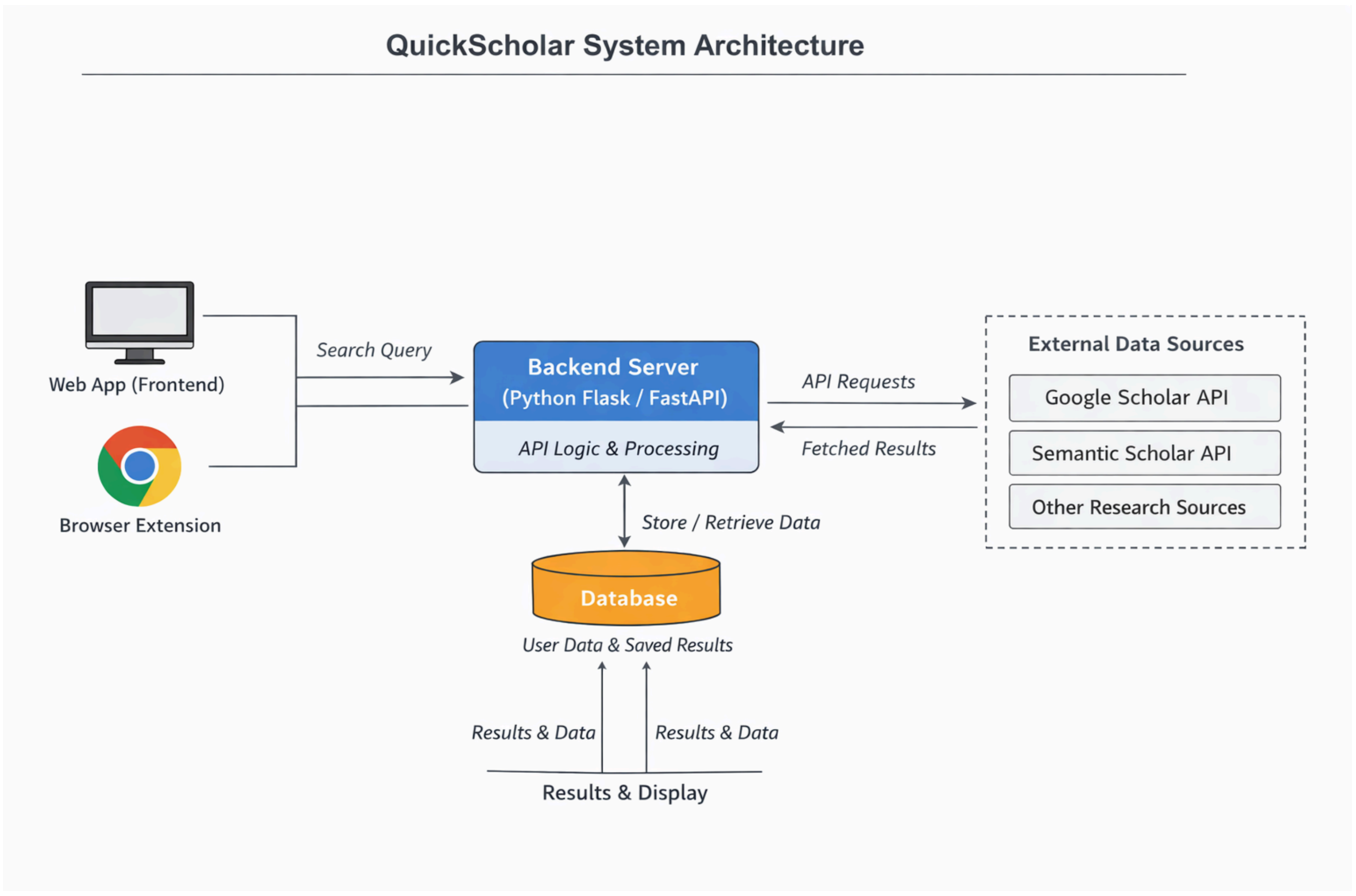
- **Backend-centric:** All logic lives in backend → easier to maintain, easier for extension to communicate
- **Frontend + Extension share backend:** No duplicate code → less complexity
- **Database optional at start:** MVP can store temporarily in memory → database added when saving/exporting is required
- **APIs first, UI second:** Search functionality must work before designing interface

4️⃣ How This Guides Your Project Creation

- You now have **clear modules**: Backend → Frontend → Extension → Database → APIs
- You know the **dependency order**:

    - Backend API & API integration
    - Database (for saving/exporting)
    - Web app frontend
    - Browser extension
- You can **estimate timelines per module**:
    - Backend search logic (3–4 weeks)
    - Database (1 week)
    - Frontend (2–3 weeks)
    - Browser extension (4–6 weeks)
- When coding, you can **tackle one module at a time**, test independently, then integrate

- Helps you **debug faster**: If results are wrong, you know the problem is backend, not frontend



DATA AND SOURCES

QuickScholar: Data & Sources Plan

Here’s how it would look for your tool (you can copy this style for your design doc):

A. Data Types

Data Field	Description
Title	Name of paper/article/report
Abstract / Summary	Short description of the work
Authors	List of authors
Year	Publication year
Source / Platform	Semantic Scholar, CrossRef, etc.
Link / DOI	Direct link to source
PDF (optional)	Link to downloadable PDF if available

B. Data Sources

Source	Access Method	Notes / Limitations
Semantic Scholar API	REST API	Free, JSON output, rate limit: 100 requests/min
CrossRef API	REST API	Free, JSON output, DOI links
Open Access journals	Scraping / API	Some require consent, check robots.txt
Other free educational platforms	API / Scraping	Only include reliable sources

---

C. Storage / Management

- **Database** (SQLite → Postgres if scaling)
    - Store: Title, Abstract, Authors, Year, Source, Link
    - Optional: Save PDF links
- **Cache layer** (optional)

- Temporarily store query results to reduce repeated API calls

---

D. How This Guides Coding

1. **Backend first**
    - Start with API integration → fetch data → parse JSON
2. **Data model**
    - Create DB schema matching your fields
3. **Search logic**
    - Backend knows exactly what fields to request, store, and return
4. **Feature design**
    - “Save/export results” can now directly use your stored fields

NON- FUNCTIONAL REQUIREMENTS

QuickScholar Non-Functional Requirements Plan

NFR Type	Clarification / Plan
Performance	- Searches should return **results within 2–3 seconds** for standard queries. - Use asynchronous calls to external APIs for faster response. - Use caching for repeated queries to reduce API calls.
Reliability	- Backend should handle failures gracefully (e.g., if one API fails, still return partial results). - Log all errors for debugging. - Database should be backed up weekly. - Minimal downtime (<1% expected during updates).
Security	- Use HTTPS for all communication. - User accounts (if added) secured with hashed passwords (bcrypt). - No sensitive data stored unnecessarily. - Protect against injection attacks if scraping or accepting user input.
Scalability	- MVP should support at least 10–50 simultaneous users. - Modular backend design allows adding more APIs or sources. - Database schema designed for easy migration to larger DB (Postgres) later.

	- Code modular for adding new features (extensions, more data fields).

4️⃣ How This Guides Coding / Project Execution

- **Performance** → Use async Python libraries like `aiohttp` for API calls

- **Reliability** → Add try/except blocks, logging, and retries for failed API calls

- **Security** → Use HTTPS, proper auth libraries, sanitize input

- **Scalability** → Modular functions, separate backend modules, DB indexes

TESTING AND QUALITY ASSURANCE

QuickScholar Testing & QA Plan

Testing Level / Type	Plan for QuickScholar
Unit Testing	- Test backend functions individually: search query parsing, API response parsing, filtering logic, save/export logic. - Use `pytest` or `unittest` in Python.
Integration Testing	- Test how backend interacts with database: saving/exporting results. - Test how frontend displays results from backend. - Test browser extension sending queries → backend → showing results.
System Testing	- Test the full user flow end-to-end: search → results → filter → save/export. - Test multiple queries in a row.
Acceptance Testing	- Verify against success criteria: • Search returns results in <3 seconds • No crashes in core user flows • Saved/exported items persist correctly • Web extension works with queries from any webpage

Performance Testing	- Run tests for multiple simultaneous users to see if response time stays acceptable. - Simulate repeated API requests to check for rate limiting issues.
Regression Testing	- After adding new features, re-run tests to ensure old features still work.
Bug Tracking & QA Process	- Use GitHub Issues to track bugs. - Log errors from backend to easily debug. - Fix high priority bugs before deployment.

4️⃣How This Guides Coding / Project Execution

1. **Write modular code**
    - Small functions are easier to test than giant scripts
2. **Test as you go**
    - Don’t wait until all features are done → test MVP features first
3. **Use automated tests for core logic**

- Especially search parser, API fetching, and filtering

4. **Manual QA for user flows**

   - Open app like a real user → click buttons, search, filter, save/export

5. **Track & fix bugs systematically**

   - Use logs, issues, or even a spreadsheet for MVP phase

DEPLOYMENT AND MAINTANANCE PLAN

QuickScholar Deployment & Maintenance Plan

Section	Plan
Hosting	- Backend: Deploy on Render, Fly.io, or Heroku (easy for Python apps) - Frontend: Serve via backend templates or deploy React frontend on Netlify / Vercel - Database: SQLite for MVP, with option to migrate to Postgres
Domain & HTTPS	- Optional: Buy domain (e.g., quickscholar.com) - Use HTTPS for secure communication
Continuous Deployment	- MVP: Manual deployment (push to GitHub → deploy to hosting platform) - Later: Setup automatic deployment from GitHub for faster updates
Monitoring & Logging	- Use backend logs to track API calls, errors, and performance - Monitor uptime (optional: free tools like UptimeRobot)

Backups & Recovery	- Backup database weekly (local or cloud) - Keep exported queries saved in user account if applicable
Maintenance Plan	- Fix bugs reported by users via GitHub Issues or email - Update APIs if sources change - Refactor code to improve performance or add new features - Maintain browser extension compatibility with latest browser versions
Security Updates	- Update dependencies regularly - Patch any security vulnerabilities in backend/frontend code

---

4️⃣ How This Guides Coding / Project Execution

1. **Design for deployment from day one**

   ○ Don’t hardcode localhost URLs → use environment variables

   ○ Prepare database connections and API keys for production

2. **Logging & error handling**

- Forces you to think about what happens if APIs fail or queries return no data

3. **Modular code**
    - Makes maintenance easier — you can fix backend, frontend, or extension independently

4. **Backup plan**
    - Ensures users don’t lose saved searches
    - Makes debugging easier

RISK AND ASSUMPTIONS

QuickScholar Risks & Assumptions

Type	Description	Mitigation / Notes
Risk: API changes / downtime	External data sources (Semantic Scholar, CrossRef) may change endpoints or block requests	Design modular API layer so you can swap APIs without breaking system
Risk: Rate limits / scraping blocks	Too many requests may get blocked	Cache results, throttle requests, use official APIs when possible
Risk: Browser extension compatibility	Browsers update, extension may break	Test with latest versions before deployment, modular extension code
Risk: Data inconsistency	Different sources may provide conflicting info	Normalize data fields and clearly display source to users
Risk: User adoption	Students may not use the tool	Start with small group (10 users), collect feedback, iterate UX

Assumption: Users want consolidated research tool	You assume students prefer one platform for research	Validate via early testing / feedback
Assumption: Free APIs are sufficient	You assume open APIs provide enough results for MVP	Have a backup plan for premium APIs if needed
Assumption: MVP features are enough	You assume search, filter, save/export are sufficient	Gather user feedback to plan next features
Assumption: Project timeline is realistic	You assume 5–6 months is enough for MVP	Track weekly progress, adjust timeline if necessary

---

4️⃣ How This Guides Coding / Project Execution

1. **Design modular code** → so you can replace or fix broken modules quickly
2. **Plan for errors** → e.g., failed API call should not crash app
3. **User feedback loop** → validate assumptions early to avoid wasted effort
4. **Schedule flexibility** → adjust timelines if risks materialize

ROADMAP AND PHASES

QuickScholar Roadmap & Phases (6-Month MVP)

Phase	Duration	Goals / Deliverables
Phase 1: Backend & API Integration	3–4 weeks	- Implement search query handling - Integrate external APIs (Semantic Scholar, CrossRef) - Parse & normalize data - Test API calls
Phase 2: Database Setup	1–2 weeks (overlaps with Phase 1)	- Design schema for saved searches & user data - Implement basic save/export functionality - Backup plan setup
Phase 3: Frontend / Web App UI	3–4 weeks	- Build search interface - Display search results with title, summary, link, authors, year - Implement filters (source, date, relevance) - Test frontend-backend integration

Phase 4: Browser Extension	4–6 weeks	- Allow users to search from any webpage - Send queries to backend - Display results in popup or redirect to web app - Test extension across Chrome, Firefox, Edge
Phase 5: UX Polishing & Testing	2–3 weeks	- Conduct usability testing with 10+ users - Improve design & flow based on feedback - Perform full QA & system testing
Phase 6: Deployment & Documentation	1–2 weeks	- Deploy web app & backend - Publish browser extension - Add HTTPS & domain if needed - Document code & usage on GitHub/LinkedIn
Phase 7: Post-MVP Iteration	Ongoing	- Fix bugs reported by users - Add enhancements based on feedback

		- Prepare for scaling or premium features

4️⃣ How This Guides Coding / Project Execution

- **Modular approach:** Backend → Database → Frontend → Extension → Testing → Deployment
- **Parallel work where possible:** e.g., database setup can overlap with API integration
- **Timeboxing:** Assign weeks to each phase to stay on schedule
- **Feedback loop:** Testing and user validation built in before final deployment

MONETIZATION IDEAS

QuickScholar Monetization Ideas

A. Pay-to-Use / Subscription Model

- **How it works:** Users pay a small monthly fee (e.g., $5–$10) for premium access.
- **What could be premium:**
    - Higher search limits per day
    - Access to more sources (premium APIs, journals)
    - Advanced filters or export formats (PDF, BibTeX, CSV)
    - Priority support / no ads
- **Why it works:** Students are willing to pay a little for tools that save them **hours of research time**.

B. Freemium Model

- **How it works:** Core features free, advanced features paid
- **Example:**
  - Free: Basic search + 3 sources + 10 results per query
  - Premium: Unlimited searches, more sources, advanced filters, saved/exported results
- **Why it works:** Lets students try before they buy → easier adoption

---

C. Ads / Sponsorships

- **How it works:** Show non-intrusive ads or sponsored results
- **Example:**
  - Banner for student tools, journals, or courses
- **Notes:** Must keep ads minimal to avoid annoying users

D. Affiliate Links

- **How it works:** If QuickScholar shows links to journals, books, or research papers, you could use affiliate links to earn a small commission
- **Example:**
  - Link to Amazon book or paid journal via affiliate
  - Earn a small cut per click or purchase
- **Why it works:** Passive revenue, doesn't disrupt core user experience

---

E. Institutional Licensing

- **How it works:** Sell QuickScholar licenses to universities, libraries, or research institutions
- **Example:**
  - University pays $X/year for access to all students

    - Admin panel for tracking usage, managing accounts

- **Why it works:** Big upfront payment → more stable revenue

---

F. Data Analytics / Insights (Optional)

- **How it works:** Aggregate anonymized data about research trends and sell insights to educational platforms or research companies

- **Caution:** Must respect privacy and GDPR/other regulations

- **Why it works:** Passive revenue, but only if user base is significant

---

3️⃣ How Monetization Guides Project Creation

1. **Pay-to-Use / Subscription**

    - Requires **user accounts and login system**

    - Requires **payment integration** (Stripe, PayPal)
    - May require **feature flag system** → show/hide premium features

2. **Freemium / Feature-based**
    - Requires **modular backend code** → free vs paid features
    - Requires **tracking user activity** to enforce limits
3. **Ads / Affiliates**
    - Requires **ad placeholders or link tracking** in frontend
    - Must ensure **UX is not ruined**
4. **Institutional Licensing**
    - Backend must support multiple accounts with admin access
    - Reporting/dashboard features for institutions

SCOPE CONTROL

QuickScholar Scope Control Plan

A. In-Scope for MVP

- Search queries via web app and browser extension
- Integrate 2–3 primary research sources (Semantic Scholar, CrossRef, etc.)
- Display search results (title, summary, authors, year, link)
- Basic filters (date, source)
- Save/export selected results
- Browser extension that triggers search from any webpage
- Deployment with basic hosting, HTTPS, and logs

B. Out-of-Scope for MVP

- Advanced AI-generated summaries or paraphrasing
- Citation formatting (BibTeX, APA, MLA) automation
- Institutional licensing features

- Ads, affiliate monetization, or analytics dashboard
- Mobile app (unless web responsive)
- Integration with premium journals

C. Scope Change Process

1. Any new feature idea is **documented**
2. Evaluate against MVP goals:
    - Does it solve the primary problem?
    - Does it fit the current timeline?
3. Decide:
    - **Include now** → adjust timeline
    - **Defer** → add to Phase 2 roadmap

    - **Reject** → keep focus

4. Update project roadmap and communicate changes (if working with a team)

4️⃣How This Guides Coding / Project Execution

- **Prioritize core modules:** Backend → Frontend → Extension → Filters → Save/Export
- **Avoid distractions:** Don’t chase “extra features” until MVP is functional
- **Maintain clarity:** Every line of code and every API integration is aligned with MVP goals
- **Facilitates iteration:** Once MVP is done and tested, scope control lets you plan Phase 2 enhancements confidently