PROBLEM MINING
FOR DEVELOPERS

How to Discover the Right SaaS Problem Before You Write Code

Deepkumar Janardhanan

PROBLEM MINING
FOR DEVELOPERS

How to find the right SaaS problem before you write a single line of code

Deep Janardhanan
DevToFounder.io

Table of Contents

Stop Hunting for Ideas. Start Mining for Problems. 5
- The actual risk nobody warns you about 5
- A different way to find opportunities 5
- What this guide covers 6

The Problem Discovery Pipeline 6
- Why a pipeline beats brainstorming 6
- What goes into the pipeline 7
- What makes something a real problem 7
- The weekly habit that makes this work 8
- A quick example of the pattern 8
- Your first goal 8

The Discovery Engines 8
- Engine 1 — GitHub Issues 9
- Engine 2 — SaaS Product Reviews 9
- Engine 3 — Job Postings 9
- Engine 4 — Developer and Industry Communities 10
- Engine 5 — Integration Gaps 10
- Engine 6 — Spreadsheet Workflows 10
- Running the engines 11

Problem Signal Taxonomy 11
- The five levels 11
  - Level 1 — Opinions 12
  - Level 2 — Complaints 12
  - Level 3 — Workarounds 12
  - Level 4 — Budget Signals 12
  - Level 5 — Mandatory Problems 12
- How to use the taxonomy in practice 13

The Idea Evaluation Matrix 13
- The eight factors 13
- Scoring and interpretation 14
- A worked example 14
- A few honest notes 15

The Question That Changes Everything 15
- What you can do right now 16
- What the full system adds 16
- The real advantage you already have 16

Your Problem Discovery Tracker 17
An example row, filled in 17
Rules that keep the tracker useful 18
Week by week 18

INTRODUCTION

Stop Hunting for Ideas. Start Mining for Problems.

Most developers who want to build a SaaS product start by asking the wrong question.

What should I build?

That question sends you inward — toward your own imagination, your own preferences, your own wishlist. It produces ideas that feel good to you but may have no relationship to what anyone else is willing to pay for. It's how talented engineers end up spending six months building something technically impressive that generates approximately zero revenue.

The question that actually works is: *what problems already exist that are painful enough to pay to solve?*

That shift — from invention to observation — changes everything downstream. Instead of defending an idea you've fallen in love with, you're evaluating evidence. Instead of hoping demand exists, you're finding proof that it already does. Instead of building and then discovering the market, you discover the market first and then build.

That's what Problem Mining is. Not a creativity exercise. A field discipline.

The actual risk nobody warns you about

When developers think about startup risk, they think about technical difficulty, scalability, infrastructure costs. Real risks, but not the ones that kill most products.

The biggest risk is choosing the wrong problem to solve.

If the problem is weak, nothing else matters. You can build excellent software around a minor inconvenience and still fail. Meanwhile, a scrappy product solving a genuinely painful problem can become a real business. That difference is decided before you write the first line of code.

Think about what the successful ones actually did. Developers were struggling to integrate payments — that frustration became Stripe. Businesses were manually connecting dozens of tools — that friction became Zapier. Startups were drowning in security audit paperwork — that pain became Vanta. In every case, the software came second. The problem came first, and it was already visible, already documented, already costing people real time and money.

A different way to find opportunities

This guide introduces a method called Problem Mining.

Instead of brainstorming ideas and hoping one sticks, you systematically observe the world for problems that already exist. Think of it like geological exploration. Valuable ore doesn't announce itself. You find it by reading the terrain, recognising patterns, and drilling where the signals are strong.

The world is full of operational problems hiding in plain sight — in GitHub issue trackers, in the "Cons" sections of G2 reviews, in job descriptions for roles that exist purely to do manually what software hasn't automated yet. Each of these sources produces signals. Some signals are weak noise. Others point directly at real economic pain.

When you collect and analyse these signals systematically, opportunities surface.

What this guide covers

This is a condensed version of the full Problem Mining system. It will show you the core methodology: how to build a discovery pipeline, which sources reliably produce strong signals, how to separate noise from real opportunity, and how to score ideas quickly so you stop wasting time on weak ones.

By the time you finish, you'll be able to generate dozens of credible SaaS candidates, filter them ruthlessly, and arrive at one or two problems actually worth building on.

The full system — including deep opportunity scoring, buyer analysis, competitive positioning, and a complete decision framework — is covered in the complete book. This guide gives you the engine. The book gives you the whole vehicle.

> *Let's start where every good product actually begins: with a problem someone already has.*

CHAPTER 1

The Problem Discovery Pipeline

Here's a habit most developers have: they encounter an interesting problem, think "that could be a startup idea," and forget about it by Thursday.

This isn't a motivation problem. It's a systems problem. Without a place to put ideas, they evaporate. And when you eventually sit down to think about what to build, your mind is mysteriously blank — even though you've stumbled across a dozen decent signals in the past month alone.

Professional founders treat opportunity discovery the way engineers treat any recurring process: they build a pipeline. A simple, low-friction system that captures signals continuously, stores them in one place, and lets patterns emerge over time.

You don't need anything fancy. A spreadsheet will do. What you need is the habit of capturing, and a structure that makes the captured data useful.

Why a pipeline beats brainstorming

Brainstorming startup ideas is a terrible way to find good ones. When you're staring at a blank page, you generate ideas from your own experience and imagination — a sample size of one. You also tend to evaluate ideas emotionally in real time, which means the first half-decent thing that comes to mind gets championed before you've seen anything better to compare it against.

A pipeline fixes both problems. It separates collection from evaluation, which is the most important discipline in early-stage opportunity discovery. You collect first, for weeks. You evaluate later, with data. That separation is what makes the process work.

It also makes patterns visible. One complaint about a broken workflow is noise. The same complaint appearing in a GitHub issue, a G2 review, three job descriptions, and a Reddit thread is a signal. You can only see that pattern if you've been collecting across multiple sources — which is exactly what the pipeline does.

What goes into the pipeline

Keep the structure simple. Six fields is enough to start:

Column	What to put here
Problem description	One sentence. What is the problem and who has it?
Source	Where you found it — GitHub, G2, job posting, Reddit, etc.
Industry	The sector affected — SaaS, fintech, healthcare, logistics, etc.
Role affected	The specific job title that experiences this problem
Signal type	Opinion / Complaint / Workaround / Budget signal / Mandatory
Evidence link	URL, screenshot, or reference — no link means no entry

> *The single most important rule: every entry needs evidence. A URL, a screenshot, a job posting link — something that proves this problem exists in the real world. Hunches don't belong in the pipeline.*

What makes something a real problem

Not every complaint is worth capturing. Before adding an entry, run it through three quick checks.

Does someone experience this regularly? A problem that shows up once a quarter is a different animal than one that disrupts a workflow every week. Recurring friction is where the commercial opportunity lives.

Does it cause measurable pain? Wasted time, operational delays, compliance risk, revenue impact — these are the kinds of consequences that motivate purchasing decisions. Mild annoyances don't.

Does it affect a specific, identifiable role? "Developers" is too broad. "DevOps engineers managing multi-cloud infrastructure" is a real buyer. The more precisely you can name who suffers from this problem, the better your chances of eventually finding and selling to them.

The weekly habit that makes this work

Set aside 30–60 minutes once a week to scan sources and add entries. During this session, you're not evaluating ideas or thinking about products. You're just collecting. The goal is to capture signals, not analyse them.

After a month of this, you'll have somewhere between 20 and 40 entries. After a few months, you'll be past 100. At that point, patterns become obvious and evaluation becomes straightforward — because you're choosing the best problem from a large, evidence-backed pool rather than rationalising whichever single idea happened to stick in your head.

A quick example of the pattern

Suppose you're casually exploring developer tooling. You find a GitHub issue in a popular monitoring repo complaining about poor API visibility. The next week, a G2 review for an API management tool mentions difficulty debugging failures in production. Then a DevOps job posting lists "build and maintain custom API monitoring scripts" as a core responsibility.

None of these individually would make you drop everything and start building. But together they're pointing at the same operational gap: teams are cobbling together manual solutions because existing tools don't give them adequate API observability. That's a pattern. That's a candidate.

That's what a pipeline reveals — not through a flash of inspiration, but through accumulation.

Your first goal

> *Set up the tracker before you do anything else. Six columns, one tab, done in under 30 minutes. Then aim to collect your first 50 entries before you start evaluating anything.*

CHAPTER 2

The Discovery Engines

Random browsing produces random results. If you want your pipeline to fill up with credible, high-signal problems rather than vague complaints, you need to know exactly where to look.

Operational pain doesn't hide. It gets documented. It gets complained about publicly. It gets written into job descriptions by hiring managers who've given up waiting for software to solve the problem. Once you know which environments reliably surface this pain, you stop waiting for inspiration and start mining systematically.

These environments are what I call Discovery Engines — specific sources where real problems appear regularly, often with enough context to understand who's affected, how badly, and what workarounds people are already using.

You don't need to run all of them. Pick two or three, spend 30 minutes each per week, and capture what you find. Over several weeks, your pipeline will fill faster than you expect.

Engine 1 — GitHub Issues

Open-source repositories are one of the richest problem sources available, and most people completely overlook them as a discovery tool.

When developers hit a limitation, they file an issue. When a workaround exists but it's painful, someone documents it. When a feature has been requested repeatedly for two years and still hasn't shipped, that unresolved thread is telling you something important about a gap the maintainers either can't or won't fill — which is often where external tools get built.

How to use it: pick popular repositories in a domain you're exploring. Sort issues by most commented, oldest unresolved, and feature requests. You're looking for frustrations that appear repeatedly — the same ask showing up from multiple users, or a workaround so elaborate it has its own wiki entry.

What to capture: requests that keep coming back, issues that have been open for months or years, and any thread where someone describes a complex manual process they've built to compensate for a missing feature.

Engine 2 — SaaS Product Reviews

G2, Capterra, and TrustRadius are goldmines that most founders treat as competitive research rather than discovery tools. They should be using them much earlier.

The "Cons" section of a product review is someone telling you, publicly and specifically, what problem their current tool fails to solve. When the same complaint shows up across dozens of reviews for multiple products in the same category, you're looking at a category-wide gap — not just one vendor's shortcoming.

How to use it: pick a SaaS category, sort by lowest ratings, and read the detailed complaint sections. Don't skim. The specific language people use to describe their frustrations is often the exact language you'll want to mirror back to them later.

What to capture: missing automation, poor reporting, integration limitations, pricing complaints that suggest the market is underserved at a certain tier. Recurring themes across multiple products in the same space are the strongest signal here.

Engine 3 — Job Postings

This one is consistently underrated, and it's my personal favourite for finding B2B opportunities.

Every job description is a company admitting, in writing, that a human being is still doing something that software hasn't automated yet. When the same manual task appears across 20 job postings for the same role — "compile weekly performance reports," "export data from multiple analytics tools," "maintain dashboards for stakeholders" — that's not a coincidence. That's a market.

How to use it: search for operational roles like marketing analyst, data analyst, operations manager, customer success manager, DevOps engineer. Skip the technical requirements section and go straight to responsibilities. Look for repetitive, manual, coordination-heavy tasks described in plain language.

What to capture: any responsibility that sounds like it belongs in software rather than a job description. The more times you see the same task across different postings, the stronger the signal.

Engine 4 — Developer and Industry Communities

Forums, Slack groups, Discord servers, Reddit threads, Hacker News — these are where people describe problems informally, often with more honesty than you'll find in a structured review.

The language is different here. Someone doesn't file a feature request on Reddit. They say "I spent three days building a script to handle this because nothing else does it properly" or "does anyone have a solution for X, I can't believe this isn't a thing yet." Those statements are pure signal.

What to capture: any post where someone describes building a workaround, expresses genuine frustration with a missing tool, or asks whether a solution exists for a specific problem. The more replies confirming the same pain, the better.

Engine 5 — Integration Gaps

Modern businesses run on 20, 30, sometimes 50 software tools simultaneously. Most of those tools don't talk to each other properly. The friction that creates is enormous, and it's often invisible to vendors who aren't looking for it.

The telltale pattern: Tool A lives in one system. Tool B lives in another. Connecting them requires a human being to download a CSV, reformat it, and upload it somewhere else. Every week. Sometimes every day. That human being is doing a job that software should be doing.

Where to find these gaps: integration marketplaces like Zapier's app directory show you what integrations exist — but the gaps are in the feature requests and community forums around those platforms. Product communities where users ask "does X integrate with Y" and the answer is "not yet, here's my workaround" are worth regular visits.

What to capture: any workflow that requires manual data transfer between two systems that clearly should be connected. The more business-critical the data, the more valuable the integration opportunity.

Engine 6 — Spreadsheet Workflows

A spreadsheet doing real operational work is almost always a sign that software doesn't exist yet for that workflow — or that the software that does exist is too expensive, too complex, or too generic to be useful.

When a team is running a business process on a spreadsheet with 15 tabs, complex formulas, manual imports from three different systems, and a person whose job is partly to maintain it —

that workflow is screaming for a product. Many successful SaaS companies started by looking at a spreadsheet a customer was using and turning it into software.

How to find them: job postings that mention Excel or Google Sheets as a required tool are a starting point. LinkedIn posts where someone shares a "free template" for managing a specific operational process often reveal workflows that haven't been productised yet.

What to capture: any operational process being managed primarily in spreadsheets, especially if it involves data pulled from multiple other systems.

Running the engines

> *Pick two or three engines. Spend 30 minutes on each during your weekly discovery session. Add signals to the pipeline. Don't evaluate — just collect. After a few weeks of consistent effort, you should have 50–100 entries and patterns will start becoming visible on their own.*

CHAPTER 3

Problem Signal Taxonomy

By now your pipeline has entries. Maybe 30, maybe 80. The problem is they don't all feel equal — and they aren't. Some are people venting on the internet. Others are companies spending real money on imperfect solutions. Treating them the same is how you end up spending three weeks investigating an "opportunity" that turns out to be ten people mildly annoyed by a UI quirk.

The Signal Taxonomy is how you tell the difference. It's a five-level ranking system that measures how much economic weight sits behind a problem. The higher the level, the more likely the problem is to produce a paying customer rather than a sympathetic nodder.

The five levels

Level	Signal type
1	Opinion — personal dissatisfaction
2	Complaint — specific product limitation
3	Workaround — users building their own solutions
4	Budget signal — companies paying to solve the problem
5	Mandatory problem — organisations must solve it

Level 1 — Opinions

"This tool is terrible." "The UI is confusing." "I wish this worked differently."

Opinions are the background radiation of the internet. Everywhere, constant, and mostly meaningless as a business signal. People complain about tools they use every day and never once consider switching. Don't discard these entries, but don't get excited about them either. An opinion only becomes interesting when it's accompanied by something from a higher level.

Level 2 — Complaints

Complaints are more specific. Missing integrations, poor reporting, performance bottlenecks, features that should exist but don't. The key limitation: complaints still don't tell you whether someone will act. What complaints do tell you is that friction exists. When the same complaint appears across many users and multiple products in the same category, you're looking at something structural — not just one vendor's failing.

Level 3 — Workarounds

This is where things get genuinely interesting.

When someone builds a workaround, they've already decided the problem is worth their time to solve. They wrote a script. They built a multi-tab spreadsheet. They duct-taped three tools together with Zapier. They're investing real effort into compensating for something that doesn't exist yet.

Workarounds are the clearest signal that a problem is real and actively disruptive — not just theoretically annoying. And when you find the same workaround appearing independently across multiple users or organisations, you have something worth investigating seriously.

Level 4 — Budget Signals

Budget signals are when organisations stop compensating for the problem themselves and start paying someone else to handle it. Consultants hired repeatedly for the same task. Agencies engaged to build custom solutions. Tools purchased that partially solve the problem even though they're overpriced or ill-fitting.

This is powerful evidence because it answers the hardest early question in SaaS: will anyone pay for this? If companies are already spending money on an imperfect solution, the answer is demonstrably yes. Your job becomes building something better and capturing budget that's already allocated.

> *A company creating a full-time role to manage a manual process is spending $60–100K per year on a problem that software might solve for $10K. That gap is a business.*

Level 5 — Mandatory Problems

The strongest signal of all. Mandatory problems exist because organisations have no choice but to solve them — regulation, legal requirement, certification process, contractual obligation. Security compliance. Financial reporting. Privacy regulations. Industry certifications. These aren't nice-to-have improvements. They're existential requirements.

This is what made Vanta possible. SOC 2 compliance was mandatory for any startup trying to sell to enterprise customers. The process was painful, manual, and expensive. The market was every SaaS company that wanted to grow upmarket. The signal strength was about as high as it gets.

How to use the taxonomy in practice

Go through your pipeline and assign a signal level to each entry. Then, for each problem, look at whether multiple signal types cluster around the same issue.

A single Level 2 complaint is background noise. A Level 2 complaint plus Level 3 workarounds plus a Level 4 budget signal pointing at the same underlying problem is a real candidate. Score them by adding the levels together — a problem with complaint (2) + workaround (3) + budget signal (4) scores a 9. Anything above 7 or 8 deserves deeper attention.

After running your pipeline through this filter, most entries will drop to low priority. A smaller set — maybe 20 to 30 out of 100 — will have enough signal weight to warrant the next step: figuring out which of those problems can actually become a product.

CHAPTER 4

The Idea Evaluation Matrix

You now have a filtered list of credible problems. Maybe 20, maybe 30. Each one has real evidence behind it. Each one passed the signal taxonomy. And now you have a new problem: they all look kind of promising, and you have no obvious way to choose between them.

This is where most developers make a mistake. They pick the idea that feels most technically interesting, or the one they've been subconsciously rooting for since they added it to the pipeline. Emotional attachment dressed up as analysis.

The Idea Evaluation Matrix exists to prevent that. It's a fast, structured scoring tool that forces you to look at eight business fundamentals for every idea — not just the ones that happen to favour your favourite. The goal isn't a final decision. The goal is to reduce 20–30 ideas to a shortlist of 8–10 that actually deserve deeper investigation.

> *It takes about 20 minutes to score your full list. That's a good trade for avoiding three months building the wrong thing.*

The eight factors

Score each factor from 1 to 5. Maximum total score is 40.

Factor	What you're evaluating
Pain severity	How badly does this problem hurt? Revenue loss, security exposure, and operational shutdowns score high. Minor inconveniences score low.
Frequency	How often does the problem occur? Daily operational headaches have more commercial gravity than annual ones.
Budget	Are organisations already spending money on this? Consultants, agencies, dedicated headcount, or overpriced partial solutions all count.
Buyer clarity	Can you clearly name who approves the purchase — not just who feels the pain? Murky buyer dynamics kill otherwise good ideas.
Distribution	Can you actually reach these people? Communities, professional networks, existing relationships, or accessible channels.
Competition	Is the market thoroughly solved, or are existing tools expensive, complex, and consistently complained about?
Founder advantage	Do you have prior experience, relationships, or domain knowledge that makes you faster to competence than a generalist?
Technical feasibility	Can a small team build a useful v1 incrementally, or does this require infrastructure scale that's hard to bootstrap?

Scoring and interpretation

Score	Interpretation
32–40	Very strong candidate — investigate deeply
25–31	Promising — worth serious attention
18–24	Weak fundamentals — proceed with caution
Below 18	Discard

A worked example

Evaluating a tool that automates support ticket classification for SaaS companies:

Factor	Score	Reasoning

Pain severity	4	Directly affects response times and customer satisfaction
Frequency	5	Happens every day, all day
Budget	3	Companies pay for support tooling, but often bundled
Buyer clarity	4	Support manager is obvious and accessible
Distribution	4	Active communities, reachable via LinkedIn
Competition	3	Some tools exist but complaints are common
Founder advantage	4	Prior experience building support tooling
Technical feasibility	4	Solid v1 achievable with a small team

> *Total: 31 — Promising opportunity worth deeper investigation.*

A few honest notes

The matrix is a forcing function, not an oracle. If you find yourself massaging scores to make a favourite idea look better, that's information. Notice it, then score honestly.

A low score on one factor doesn't automatically kill an idea. A mandatory compliance problem might score low on frequency but extremely high on pain severity, budget, and buyer clarity — and still be a strong opportunity. The matrix helps you see the full picture, not just the parts you're already excited about.

What you're looking for is a shortlist of ideas where the business fundamentals are solid — where you can clearly name who buys it, why they care, how you'd reach them, and why existing solutions fall short.

CONCLUSION

The Question That Changes Everything

Most developers who want to build a SaaS product start by asking the wrong question.

"What should I build?"

That question sends you inward — toward your own imagination, your own preferences, your own wishlist. It produces ideas that feel good to you but may have no relationship to what anyone else is willing to pay for.

The question that actually works is: *"What problems already exist that are painful enough to pay to solve?"*

That shift — from invention to observation — changes everything downstream. Instead of defending an idea you've fallen in love with, you're evaluating evidence. Instead of hoping demand exists, you're finding proof that it already does.

What you can do right now

This week: set up the discovery tracker. Six columns, one tab, 20 minutes. Done.

Next week: pick two discovery engines and run them for 30 minutes each. Add every signal that passes the basic test.

Over the following weeks: keep collecting. Don't evaluate yet. Let the database grow until patterns start becoming obvious on their own.

After 4–6 weeks: run your entries through the Signal Taxonomy. Score them. Watch most drop away and a smaller set rise to the top.

Then: apply the Evaluation Matrix to your top candidates. Score all eight factors honestly. Sort by total. Your shortlist will be right in front of you.

> *At that point you'll have something most aspiring SaaS founders never have: a ranked list of real opportunities, each backed by real evidence, evaluated against consistent business criteria. That's not a starting point. That's already further than most people get.*

What the full system adds

This guide covered the discovery and filtering end of Problem Mining. The complete book picks up where this ends.

It covers the Opportunity Scorecard — a deeper evaluation that examines long-term defensibility, pricing power, switching costs, and moat potential, so you're not just picking the most visible problem but the one with the strongest structural foundation for a durable business.

It covers buyer mapping — the discipline of identifying not just who uses the product but who approves the purchase, why they care, and what language they use when they justify the spend internally. In B2B SaaS, this distinction is often what separates products people love from products companies actually buy.

It covers competitive positioning — not as a slide deck exercise but as a practical tool for finding the specific gap your product occupies in a market that already has solutions, which is most markets worth entering.

And it covers the final selection framework: the process that takes your shortlist and produces a single clearly chosen problem — the one you build on with confidence.

The real advantage you already have

Developers are unusually well-positioned for this. You're already close to the systems where problems surface. You read GitHub issues naturally. You understand what a workaround means and how much effort it represents. You can recognise a manual process that shouldn't be manual. You have the technical credibility to talk to potential customers and be taken seriously.

Most non-technical founders have to learn the landscape you already live in. You're not starting from zero — you're starting with a significant information advantage. Problem Mining is just the framework for using it deliberately.

> *The developers who build successful products aren't the ones who had better ideas. They're the ones who found better problems. Now you know how to find them.*

APPENDIX

Your Problem Discovery Tracker

This is the only tool you need to start. Set it up before you do anything else.

Create a spreadsheet — Google Sheets, Excel, Notion, Airtable, whatever you already use. Add these columns:

Column	What to put here
Problem ID	P001, P002, P003 — just a unique reference
Problem description	One sentence. What is the problem and who has it?
Source	Where you found it — GitHub, G2, job posting, Reddit, etc.
Industry	The sector affected — SaaS, fintech, healthcare, logistics, etc.
Role affected	The specific job title that experiences this problem
Signal type	Opinion / Complaint / Workaround / Budget signal / Mandatory
Signal score	1–5 matching the taxonomy from Chapter 3
Evidence link	URL, screenshot, or reference — no link means no entry
Notes	Workarounds observed, related problems, anything useful

An example row, filled in

Field	Example
Problem ID	P001
Problem description	Marketing analysts manually export data from three analytics tools weekly to compile performance reports for leadership
Source	Job posting — LinkedIn
Industry	B2B SaaS
Role affected	Marketing analyst
Signal type	Workaround
Signal score	3
Evidence link	linkedin.com/jobs/[posting URL]
Notes	Same task described in 6 other job postings. Spreadsheet template shared in a marketing ops Slack group for doing this manually.

Rules that keep the tracker useful

No evidence, no entry. If you can't attach a link or reference, the problem doesn't go in. Hunches belong in a notes file, not a discovery pipeline.

One sentence descriptions only. If you can't describe the problem in one sentence, you don't understand it well enough yet. Keep browsing.

Score at capture time. Assign the signal type and score when you add the entry, while the context is fresh. Retroactive scoring is less accurate and takes longer.

Don't evaluate while collecting. The moment you start asking "but could this be a product?" during a collection session, you've stopped collecting. Keep the two activities completely separate.

Week by week

Weeks 1–4: collect only. Run two or three discovery engines per week, add everything that passes the basic test. Aim for 10–15 new entries per session.

Weeks 4–6: start scoring. Go through your entries and apply the Signal Taxonomy. Anything scoring 7 or above across combined signals gets flagged for deeper attention.

Week 6 onwards: run the Evaluation Matrix against your flagged entries. Your shortlist emerges from the top scores.

> *By week 8, most people who follow this process consistently have 60–120 entries, a clear cluster of 15–20 credible problems, and 6–10 strong candidates ready for serious evaluation. That's not a mood board. That's a pipeline with real data in it.*

Problem Mining for Developers

The complete system is available at

devtofounder.io