2025-2026

THE FOUNDERS TECH TOOLKIT

ESSENTIAL SYSTEMS AND TOOLS FOR BUILDING SCALABLE STARTUPS

FOR FOUNDERS WHO WANT TO SCALE FAST BUT WITHOUT CUTTING CORNERS

ALBERTO ZUIN

Contents

Introduction

This toolkit was created to help founders build MVPs that move fast, without breaking later. A common mistake I see is confusing a PoC with an MVP. While a PoC explores technical feasibility, an MVP is a user-facing product with a solid foundation to support real usage, feedback, and growth.

In this guide, we'll focus on building lean but robust MVPs that are fast to launch, secure by design, and scalable enough to grow with you. Consider this a mentor's voice beside you as you confidently shape your product.

In today's environment, speed is critical — but so is trust. Founders are under pressure to ship quickly, impress investors, and iterate fast. But I've seen startups fail at both ends of the spectrum: some get stuck trying to perfect an overengineered product that never launches, while others move fast with a fragile, throwaway tech stack that can't support real growth. The hard truth? Neither extreme works. This guide — and my role as a technical mentor — is about finding that healthy middle ground: moving fast with purpose, making smart trade-offs, and building just enough foundation to support what comes next.

This isn't a theoretical playbook. It's shaped by hands-on experience building and scaling products in high-pressure environments — and occasionally fixing mistakes that could have been avoided with a clearer foundation. Whether you're a solo founder or have a small tech team, the goal is to give you just enough structure to move fast — without technical debt becoming your legacy.

You don't need to over-engineer or build like a Big Tech company. But you do need to make smart early decisions — about architecture, security, team workflows, and product scope. That's where this guide comes in: helping you focus on what really matters at the MVP stage.

This toolkit is especially for first-time founders, technical advisors, or small teams without a dedicated CTO. If you're figuring out product–market fit while juggling deadlines and limited resources, this guide will help you stay focused and make scalable, defensible choices.

You'll find short chapters, practical checklists, and cheat sheets — no jargon, no fluff. It's designed so you can dip in quickly when making key decisions or share with your team to stay aligned.

My name is Alberto Zuin, and over the past 25+ years I've helped startups and scaleups architect production-grade systems that are both investor-ready and founder-friendly. For each section, I'll include a short story — a real scenario from my past work — to give you context. Sometimes we got it right, sometimes we didn't, but every story comes with a lesson that can help you avoid common traps.

Let's get started.

Section 1: Start Small, Scale Smart

Choosing the Right Architecture for Your Phase

My Story

I started my career deep in the world of IT systems engineering, configuring routers and switches and building internet services on bare-metal servers. In those days, infrastructure *was* the product. Then came virtualisation, and with it, a wave of new possibilities.

I was an early adopter of **OpenNebula**, and one of my first ventures — the original version of **MOYD** — was a startup that built a dynamic DNS service tailored for OpenNebula and OpenStack deployments. That project was later acquired by Italy's largest ISP to support their infrastructure. It was a pivotal moment for me and the evolution of what we now call "cloud-native."

Since then, I've followed — and often adopted — every major shift: from building private IaaS clouds, embracing containers, and running workloads on modern PaaS platforms and serverless (FaaS) functions. The reality I've learned through this journey is simple:

The more "bare" the service, the lower the platform cost — but the higher the maintenance and complexity. The more complete the service (PaaS, FaaS), the higher the cost, but the more freedom founders have to focus on the product.

Before talking about stacks, we need to talk about context.

Not every startup needs Kubernetes, and not every MVP should run on AWS. The best architecture allows you to move fast, learn from users, and grow without rewriting everything in six months.

Founders often ask:
"What should we build on?"

My answer:
It depends on how quickly you need to ship, how much control you need, and who's on your team.

Let's explore three common levels of architecture and what trade-offs they bring.

1. Platform-as-a-Service (PaaS)

e.g., Heroku, Vercel, Firebase

Best for: Solo founders, very early teams, fast MVPs.

Why it works:

- Zero DevOps: just push code
- Autoscaling, CI/CD, and monitoring built in
- Ideal for getting user feedback ASAP

Trade-offs:

- Limited flexibility and control
- Scaling costs can spike
- “Outgrowing” the platform may require migration

Use it if your main risk is “Will anyone use this?” not " How do we handle 10,000 users?”

2. Containerised App Hosting

e.g., AWS ECS, Google Cloud Run, Railway

Best for: Teams with moderate technical resources

Why it works:

- More control over environments and deployments
- Easier to adopt best practices (e.g., IaC, secrets management)
- Can scale further before re-architecture

Trade-offs:

- Some DevOps overhead
- Requires container knowledge (Docker)
- Still needs cloud networking basics

Use it when you need to balance speed, scalability, and cost, and have someone comfortable with basic infrastructure.

3. Infrastructure-as-a-Service (IaaS)

e.g., EC2, Load Balancers, VPCs

Best for: Scale-ups, regulated industries, or complex architectures

Why it works:

- Total control and flexibility
- Mature security and compliance support
- Multi-service orchestration is possible

Trade-offs:

- High complexity and cost
- DevOps/infrastructure expertise required
- Slower time-to-market for MVPs

Use it only if you have in-house infrastructure skills or your product requires tight infrastructure control from day one (e.g., fintech, health tech).

Honourable Mention: Function-as-a-Service (FaaS)

e.g., AWS Lambda, Google Cloud Functions, Vercel Functions

Best for: Lightweight, event-driven tasks; prototypes; backend microservices

Why it works:
- No infrastructure to manage at all
- Extremely cost-effective at small scale
- Automatically scales with usage
- Great for simple APIs, background jobs, or glue logic

Trade-offs:
- Not suitable for complex applications or long-running processes
- Cold starts can affect performance
- Limited execution time and memory
- Can lead to fragmented architecture if overused

FaaS is technically a subcategory of PaaS, but it's worth calling out: you're not hosting an app, you're hosting functions. If all you need is a webhook, a simple backend endpoint, or a daily cronjob, FaaS can be the fastest and cheapest way to get moving.

So... Where Should You Start?

If you're a typical early-stage startup, PaaS gets you to users fastest.
If you're aiming for investment or handling sensitive data, container-based hosting on a major cloud provider (e.g. AWS Fargate) offers a good middle ground.
And unless you're building a bank or data platform, full IaaS can wait.

Criteria	PaaS (Heroku, Vercel)	Containers (ECS, Railway)	IaaS (EC2, Load Balancers)
DevOps Required	None	Moderate (Docker, IaC)	High (networking, IAM, logging, etc.)
Customization/ Control	Limited	Good	Full control
Scalability (up to Series A)	Limited to Moderate	Good	Excellent
Security & Compliance	Basic (shared responsibility)	Stronger (cloud native)	(with effort)
Ideal For	Solo founders, MVPs	Growing startups, light ops teams	Scale-ups, regulated industries
Cost Complexity	Predictable but can spike	Efficient with tuning	Can get expensive fast
Speed to Launch	★★★★ Very fast	★★★ Moderate	★ Slow

Relational, NoSQL, or Object Storage? It Depends on What You're Building.

Not all data is created equal, and not all apps need the same database.
Choosing the wrong one won't just slow you down; it can force major rewrites later.

Let's compare the three main categories used by startups, and when to choose each:

Data Storage Options at a Glance

	Relational (SQL)	**NoSQL**	**Object Storage (e.g. S3)**
Examples	PostgreSQL, MySQL, SQLite	MongoDB, DynamoDB, Firestore	AWS S3, GCS, Cloudflare R2
Best For	Structured, relational data	Unstructured or flexible schemas	Files, media, backups, logs
Schema	Strict, normalized	Flexible or schemaless	No schema
Consistency	Strong ACID compliance	Eventual or configurable consistency	No consistency guarantees (file-level)
Scaling	Vertical, sharding at later stage	Horizontal scaling out of the box	Infinitely scalable
Querying	Powerful joins, transactions	Fast key/value or document lookups	No querying, just file paths/URLs
Dev Effort	Moderate (ORMs help)	Easier at first, can become messy	Minimal
Use Cases	Dashboards, accounts, payments	Chat apps, IoT, user preferences	Images, PDFs, audio, exports

When to Use Each

Relational DB (SQL) — the Safe Default

Use this if you're building **anything transactional**: users, payments, dashboards, reports.
PostgreSQL is my go-to: it is stable, battle-tested, and well supported by every backend framework and ORM.

Choose when you care about **data integrity**, relationships, and queries.

NoSQL — When Flexibility Wins

Use this when your data has no fixed structure, or your access patterns are very simple (e.g., ID → value lookups).
MongoDB is great for fast prototyping. DynamoDB can scale massively, but requires deeper AWS knowledge.

Choose when you need **speed, simplicity, or schema freedom,** and you'll know how to access your data.

Object Storage — For Files, Not Data

Use S3 or similar to store **images, documents, audio**, or even **JSON blobs** for Machin Learning use cases.
Don't use it like a database — it won't scale that way.

Choose when you need to **store and retrieve blobs**, not query structured data.

Recommendation for MVPs

In most MVPs:

- Use **PostgreSQL** for structured app data (users, activity, settings)
- Use **S3** (or compatible) for file uploads
- Avoid mixing NoSQL unless your product truly needs it (e.g., flexible schemas or massive scaling early)

My Story

In one of my previous projects, the biggest one in my career, I chose **MySQL** over my usual preference for **PostgreSQL**.

The reason?
We needed to **disseminate data across geographically distributed data centres. At the time, MySQL's native replication offered a simpler path to multi-region syncing,** something PostgreSQL didn't support as cleanly out of the box.

This was **before widespread cloud adoption**. Today, the same challenge would be solved with **cloud-native replication features** like:

- **Aurora Global Databases** (AWS)
- **Cloud SQL with read replicas** (GCP)
- **PlanetScale's multi-region MySQL**

👉 **Lesson:**
Sometimes, your stack choice is shaped more by infrastructure maturity and deployment constraints than by developer preference.

Always ask: "What problem are we solving — and what trade-offs matter right now?"

CI/CD and Infrastructure-as-Code (IaC)

Speed isn't just about writing code fast — it's about reliably getting it in front of users.
That's where **CI/CD** and **IaC** come in: even at the MVP stage, they help you move faster *and* safer.

Let's start with the basics, then answer a key question:
"Do I need all this if I'm just using Heroku?"

CI/CD: Continuous Integration & Delivery

What It Does:

- Automatically runs tests and checks on every code change
- Builds and deploys your app without manual steps
- Reduces "it works on my machine" problems

Minimal Setup for MVPs:

- Use **GitHub Actions** or **GitLab CI**
- Trigger on main or release branch push
- Lint, test, deploy — that's enough

Even for Heroku or Vercel, this adds guardrails and saves time.

IaC: Infrastructure as Code

What It Does:

- Defines your infrastructure (servers, buckets, secrets, etc.) in version-controlled code
- Enables reproducibility and disaster recovery
- Prevents undocumented, manual changes

When It's Useful:

- If you're using **cloud-native resources** (like AWS S3, RDS, ECS)
- If your infrastructure has more than 2–3 moving parts
- If you want to avoid "clickops" (configuring everything manually in a dashboard)

Tools like **Terraform (OpenTofu)**, **Pulumi**, or even **CloudFormation** make this manageable, especially when versioned with your app.

What If I'm Using Heroku or Vercel?

The good news is that these platforms abstract most of the infrastructure complexity. **You don't need a full-blown IaC**.

But that doesn't mean you skip automation entirely.

My Suggestion:

- Still use **CI/CD** to lint/test/deploy
- Document key platform config (e.g., env vars, add-ons) in a .md file or JSON for consistency
- Consider light IaC *only* if:
    - You're mixing in AWS/GCP services (e.g., S3, RDS)
    - You plan to outgrow the PaaS within 6–12 months
    - You're sharing the infrastructure setup across a team

My story

In my projects, even for simple apps, I've always introduced lightweight CI/CD from day one. And for anything beyond pure Heroku/Vercel, I've used **Terraform** — not because it's trendy, but because I've learned this the hard way:

Infra you can't version is infra you'll forget.

When teams grow or infrastructure evolves, your future self (or your future engineer) will thank you for having the basics in place: a repo for infrastructure, and a pipeline that deploys without manual fiddling.

This naturally leads us to GitOps, a methodology that treats *everything* (code, infra, configuration) as version-controlled, declarative, and automatically reconciled.

What is GitOps (and Why Startups Like It)?

GitOps is the idea that Git becomes the single source of truth, not just for application code, but also for infrastructure and deployment state. Instead of "clicking deploy" or manually tweaking cloud settings, everything flows from your Git repository. In practice, it means:

- Every change (infra or app) is made via a pull request
- A CI/CD pipeline or controller syncs changes automatically
- Rollbacks are as easy as reverting a commit

Why it works for startups:

- **Auditability:** You always know who changed what and when
- **Speed + Consistency:** One pipeline to rule them all, no manual drift
- **Collaboration:** Engineers speak the same language, Git, not GUI
- **Disaster recovery:** Git becomes your backup of infrastructure state

But Be Careful…

Despite the appeal, GitOps can become **overkill** for early-stage teams if you don't need all that rigour yet.

- You'll spend time writing and debugging YAML when you could be building a product
- It introduces tools and workflows (ArgoCD, Flux, custom controllers) that come with steep learning curves
- Without discipline, it leads to fragile automation: "it worked locally, but broke the pipeline"

Use GitOps as a mindset, not dogma.
Start with versioning infrastructure and automating deployments: that alone gives you 80% of the benefit. Full GitOps can come later, once your team or product complexity justifies it.

Picking the Right Development Stack

Frameworks Shape Speed, Team Fit, and Future Hiring — Choose Wisely.

Backend Frameworks for Web APIs

Language / Framework	Pros	Cons
Python – FastAPI	Async-ready, fast to build, great docs, easy to learn	Less batteries-included than Django
Python – Django	Full-stack, ORM, admin panel, mature ecosystem	Monolithic feel, heavier than FastAPI
Ruby – Rails	Convention over configuration, ideal for rapid MVPs	Can become opinionated and heavy with scale
Ruby – Sinatra	Lightweight, great for microservices or quick APIs	Minimal — lacks built-in tools (ORM, validations, etc.)
C# – ASP.NET Core	High performance, great tooling (esp. in Microsoft environments)	Learning curve, better suited to larger teams or Windows-heavy orgs
Node.js – Express	Fast, minimal, JavaScript everywhere (frontend/backend)	Minimal out of the box, needs careful structure
Node.js – NestJS	TypeScript-native, modular, structured like Angular	Learning curve if new to TypeScript or backend dev
Go – Fiber/Gin	Extremely fast, good for high-performance APIs	More low-level, not ideal for quick MVPs unless you know Go

My usual picks:

- **FastAPI**: when AI, async, or Python-based tools are in play
- **Rails**: when you need to move fast and build full features quickly
- **NestJS**: when the team is already working with TypeScript

Frontend Frameworks for Web Apps

Framework	Pros	Cons
React	Huge ecosystem, flexible, supported by Vercel (Next.js)	Needs careful state management (can get messy)
Vue.js	Great docs, beginner-friendly, batteries included	Smaller ecosystem than React; hard to hire for in some regions
Angular	Enterprise-grade, structured, batteries-included	Steep learning curve, heavier setup
Svelte	Extremely lightweight, reactive by default	Smaller community, less mature than React/Vue

My take:

- **React** is the default for flexibility and hiring
- **Vue** is great for small teams with limited experience
- **Svelte** is fun and fast, but consider long-term support

Single App vs Split Frontend/Backend

Architecture	Pros	Cons
Monolithic App	Easier to deploy, fewer moving parts, ideal for MVPs	Harder to scale parts independently later
Split Frontend/Backend	Scalable, better for teams, more secure via API gateway	More infra complexity, needs CI/CD pipelines, CORS headaches

Advice:

- Start monolithic (e.g., Rails or Django full-stack) **if speed is king and the product is simple**
- Split FE/BE **if your team has frontend/backend specialists** or if you're building mobile apps too
- Either way, define APIs cleanly from day one — even if it's just a module boundary

My Story

Over the years, I've worked with all sorts of stacks — from monoliths to microservices, and from legacy PHP to serverless APIs. What I've learned is that **tech choices aren't just technical — they shape how fast you move, how well your team collaborates, and how painful things become when you grow**.

In the very first version of MOYD, I built the backend in **PHP with Symfony** and paired it with a frontend written in **Angular**. It did the job, and we got the product working — but even then, the tight coupling between frontend and backend made each update a bit more painful than it should have been. Still, it was the right choice then, and it proved enough to attract interest and ultimately have it acquired.

At ColossusBets, we launched the initial PoC using **Rails**, and it helped us move incredibly fast. However, the system was more complex than a typical MVP: multiple frontends, integrations with payment gateways and external partners, and eventually, a growing team with dedicated frontend engineers. Midway through the MVP phase (about two months in), we made the call to **split the frontend and backend**. That decision wasn't just about scale but about working efficiently as a team. Once the APIs and environments were properly set up, we accelerated again.

In some **recent startups I've helped**, I leaned toward **FastAPI backends hosted on AWS Lambda**, paired with whatever frontend the team was most comfortable with — sometimes

React, sometimes even no UI at all at the start. This setup allowed async API calls, quick iteration, and minimal DevOps overhead, perfect for small teams validating ideas.

My takeaway? There's no perfect stack, just the one that fits your team, timeline, and rework tolerance. Start with what gets you learning, not what gets you stuck.

Use What Already Works

You're not launching a login system. Or a file uploader. Or a dashboard framework.

You're building a product and you need to get it to market before your funding, motivation, or window runs out.
That means using **battle-tested services** wherever possible. Skip the ego, skip the "custom everything" mindset.

Here are **some plug-and-play services** I recommend across different parts of the stack:

Recommended External Services for MVP Speed

Category	**Tool / Service**	**Why Use It**	**Free Tier / Startup Perk**
Authentication	Auth0, Clerk, Firebase Auth	OAuth2, social login, magic links	Generous free tier (Auth0: 7k users, Clerk: 5k)
Payments	Stripe, Paddle, Lemon Squeezy	Billing, tax, invoicing	Stripe Atlas / Startup program via partners
File Uploads	Filestack, Uploadcare, Cloudinary	CDN, resizing, optimization	Free plans (Cloudinary: 25 credits/month)
Transactional Email	Postmark, Resend, SendGrid	Reliable email delivery + templating	Resend: 100 emails/day, SendGrid: 100/day
Notifications	OneSignal, Courier, Pusher	Push, SMS, in-app	OneSignal: Free up to 10k subscribers
Logging & Errors	Sentry, LogRocket, Datadog	Exception tracking, session replays	Sentry: 5k events/mo free; Datadog: free trials
Monitoring	UptimeRobot, Better Stack, Checkly	Status checks, alerts	Better Stack: Free Dev plan; UptimeRobot: free tier
Forms & Surveys	Tally, Typeform, Fillout	Collect feedback or lead info	Tally: unlimited forms for free
CMS / Content	Sanity, Strapi, Prismic	Content management for non-devs	Sanity: generous free quota; others via GitHub
Search	Algolia, Meilisearch, Typesense	Fast, typo-tolerant search	Algolia: 10k ops/mo; Meilisearch: open source
Data Sync	Supabase Realtime, Ably	WebSockets & live updates	Supabase: Free tier for small projects
PDF Generation	PDFShift, DocSpring, Placid	Invoice or quote generation	DocSpring: free docs/month; Placid: 100 renders/mo
Dashboards & BI	Retool, Metabase, PowerBI	Build internal tools fast	Retool Free Tier for dev use; Metabase is open source

Pro Tip:

***If it's not your core value**, buy it.*
Even if you replace it later, you'll move 10× faster now — and that speed is your real moat.

Tip for Founders:

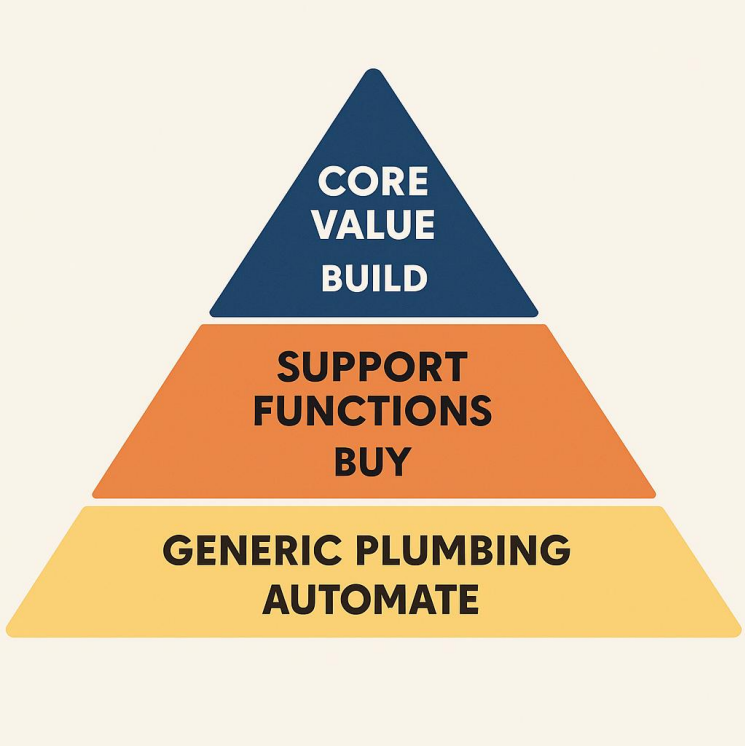
*Always check if these tools offer **GitHub Student/Startup perks**, or apply via platforms like AWS Activate and Stripe Atlas Benefits. You can save thousands.*

My story

You need to know what's **foundational** to your product — and what isn't.

I've built **email servers for over a decade**, including complex setups like Italy's **PEC-certified systems**, which require encryption, tracking, and delivery guarantees. But unless your product is an email provider, I'd never recommend running your mail server today. Use Postmark, Resend, or SendGrid: they are cheap, reliable, and battle-tested.

But there *are* exceptions — and I've lived them too.



In the first version of MOYD, we started with **PowerDNS** as a quick way to prove our idea: dynamic DNS provisioning for virtual machines. It worked fine for a PoC, even if it wasn't blazing fast. However, once we validated that DNS was our core differentiator, we rebuilt the system from scratch: a high-performance Scala DNS server optimised with **in-memory data** and **database sync via callbacks**. That version became the fastest solution on the market, surpassing rock-solid systems like Bind and ultimately led to our acquisition.

The lesson? Reuse everything you can — unless it's what makes you different.

Startups that just string together a few SaaS tools via Zapier rarely stand out.
Invest your time and code in what gives users a reason to care. Buy the rest.

Section 2: Dev Process for Speed & Quality

How to Stay Organized Without Slowing Down

You don’t need a Scrum Master to ship fast, but you do need focus, visibility, and just enough structure to avoid chaos. Agile isn’t about sticking religiously to rituals but adapting to deliver value *quickly and consistently*.

Both **PMI (Project Management Institute)** and **PeopleCert (PRINCE2 Agile)** highlight this flexibility. PRINCE2 Agile emphasises that Agile frameworks should be tailored to the organisation’s **size, culture, and delivery environment**. PMI's **Agile Practice Guide** also makes it clear: “Agile approaches are not a one-size-fits-all solution — they are a spectrum of practices to be adapted.”

So if a Project Manager insists that every team must run full standups, burndowns, retros, sprint reviews, and velocity reports — regardless of team size or maturity — they’re missing the point.

A team of two doesn’t need a process designed for a team of twenty. The process must **fit the reality**, not force the team into a rigid mould.

Small Teams Are Still Agile

Agile guidelines often talk about **cross-functional teams of 6–12 people**, including roles like:

- Product Manager
- Designer
- Developers
- QA Engineer
- DevOps/Infrastructure

If you're just starting with 1–2 developers, don't feel like you’re “doing it wrong.” Early-stage teams are *meant* to be small and nimble. The key is to keep feedback loops short, stay aligned on priorities, and **deliver working software weekly**.

Agile for 2-Person Teams

- Use **weekly planning calls** or async check-ins (e.g. ClickUp, Notion, Trello)
- Track work in **Kanban**: To Do → In Progress → Done → Blocked
- Keep it visual, lightweight, and focused on **deliverables**
- Don’t obsess over velocity — obsess over **user feedback and value**

Tools That Work at This Stage

Need	Suggested Tool	Why It Works
Task tracking	ClickUp, Linear, Trello	Simple, visual, flexible
Git + CI/CD	GitLab, GitHub Actions	Automate deploys, tests, and previews
Team sync	Slack, Discord, Telegram	Async-friendly for small, fast-moving teams
Documentation	Git repo + Markdown, Notion	Don't overthink it — just write what matters

CI/CD: Automate the Boring Stuff Early

Even in tiny teams, automating testing and deployment avoids time-wasting rituals and late-night mistakes.

Start With:

- Push to the "main" branch triggers tests
- Merge to release or tag deploys to staging/prod
- Add basic linters and test runners

Rule of thumb: If you repeat something more than twice, script or automate it.

When to Start Testing Automation

Start **light and early** — don't wait for scale.

MVP Testing Priorities:

- **Unit tests** for key logic (e.g., calculations, validations)
- **E2E tests** for signup, login, and payment flows (use Playwright or Cypress)
- **Smoke test** after deploys (basic "does it load?" check)

*Don't aim for 100% coverage — aim for **confidence** in core flows.*

Final Advice:

The best process is the one that helps you:

- Ship weekly
- Sleep at night
- Learn from users

If your board isn't moving, your product won't either.

A Week in the Life of a 2-Person Dev Team

Two people. One goal. Clear priorities. No fluff.

Day 0 (Sunday or Monday morning)

Sprint Planning (~30 minutes)

- Review what was done last week
- Pick 3–5 priorities tied to a goal (e.g., “Let’s launch signup flow”)
- Break into tasks (ClickUp board: To Do → In Progress → Done)
- Assign responsibilities

Example outcome:

Goal: Enable user registration
Tasks: Design signup page, build API endpoint, add validation, test in staging

Day 1–3 (Monday–Wednesday)

Deep Work Days

- Focus on 1–2 features or technical spikes
- Commit early, push often
- Use GitLab CI to deploy preview/staging versions automatically
- Update ClickUp board daily or asynchronously (no standups needed)

Tip: Work in **feature branches**, merge via PR with a short review (even solo — future-you will thank you)

Day 4 (Thursday)

QA + UAT (User Acceptance Testing)

- Run smoke tests (automated or manual)
- Try flows in staging (use dummy accounts)
- Fix obvious bugs
- If possible: demo to a founder, user, or mentor for feedback

Day 5 (Friday)

Deploy to Production + Retro

- Tag release, deploy with CI/CD
- Confirm monitoring/logs are working
- Optional: short async retro (What went well? What was painful?)

Ask:

“Did we ship something users can see or benefit from this week?”
“Did we learn something that changes what we build next?”

Weekend Bonus (Optional):

- Respond to any prod issues
- Sketch ideas for the next sprint
- Rest — you’re building a startup, not burning out

Final Tip:

A good sprint ends with working software.
If you’re planning more than you’re shipping, simplify.

My story

At ColossusBets, we’ve been lucky to work with a **Project Manager who truly understands Agile as a flexible mindset**. Unlike many traditional PMs, he didn't force a textbook process. Instead, he adapted our workflow to suit a challenging environment:

- We had **multiple concurrent projects**, often run by the same developers
- We depended heavily on **external integrations**: KYC providers, payment gateways, regulators, B2B partners, many of which became blockers outside our control
- Our priorities could shift weekly based on compliance updates or market opportunities

What made it work wasn’t the ceremony, but the collaboration and flexibility. Tasks were visualised in a shared board, blockers were surfaced early, and delivery happened in short, focused cycles.

It’s also worth noting that our PM was responsible only for the technical development, not the entire commercial project. This separation helped keep things lean and focused.

To streamline the process even further, we used GitLab’s built-in project management features and found them incredibly effective. Linking issues, commits and merge requests, and CI/CD pipelines in a single platform removed friction, improved traceability, and reduced the need for duplicating status across tools.

A similar setup is possible in GitHub Projects, which now supports linked PRs, automation, and Kanban-like boards.

If your team already lives in GitLab or GitHub, using their native tools instead of a dedicated external system like Jira can simplify your workflow and speed delivery, especially for small teams.

This experience taught me that the right PM doesn't enforce a framework — they enable delivery by adapting it to the team's real-world constraints and tools.

Who Should Manage Integration-Heavy Projects?

When a project relies heavily on **external parties**, it's less about code and more about **orchestration**. The best person to manage these projects is often a **hybrid Product/Project Manager** — someone who:

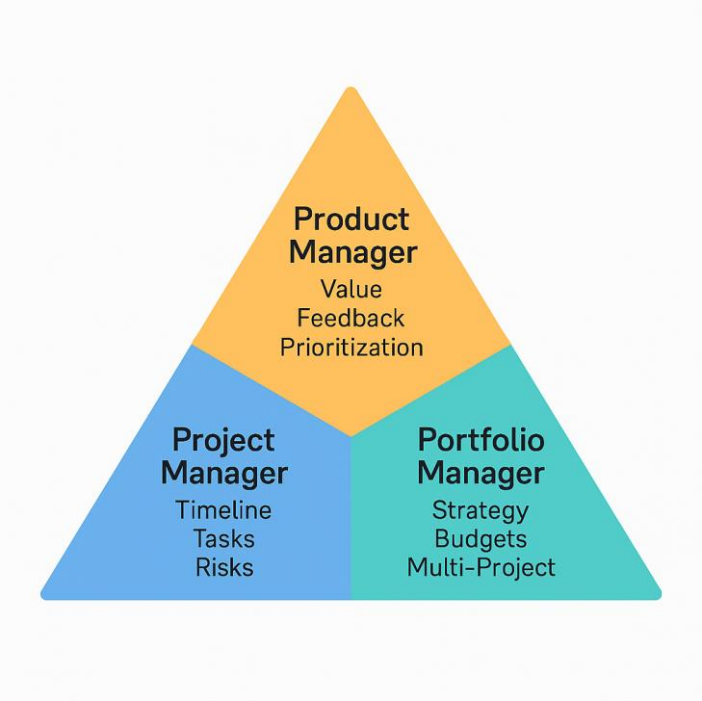
- Understands the *technical flow*
- Can manage third-party relationships and SLAs
- Tracks external dependencies proactively
- Communicates uncertainty to stakeholders

Sometimes, this person comes from a **Technical Account Manager**, **Solutions Architect**, or **Delivery Manager** background. Whatever the title, the key is that they blend **product understanding, project discipline, and external communication**.

To scale well, it helps to understand the difference between typical leadership roles, especially if you're hiring or planning for growth:

Role	**Focus**	**Typical in MVP?**	**Notes**
Project Manager	Timeline, tasks, risks, delivery	Often overkill	Best for complex, deadline-driven delivery (e.g., compliance, clients)
Product Manager	Value, feedback, prioritisation	Essential	The voice of the user and market, critical even in small teams
Portfolio Manager	Strategy, budgets, multi-project alignment	Not needed yet	Relevant only in multi-product orgs or late-stage

If you’re building an MVP with two engineers, a Project Manager just to manage a board is likely a waste. But a Product Manager — even part-time — helps ensure what you build **matters to users**.



Section 3: AI Integration Essentials

Building with AI as a Layer — Not as a Gimmick

Forget building your own model: that's not your startup's job.

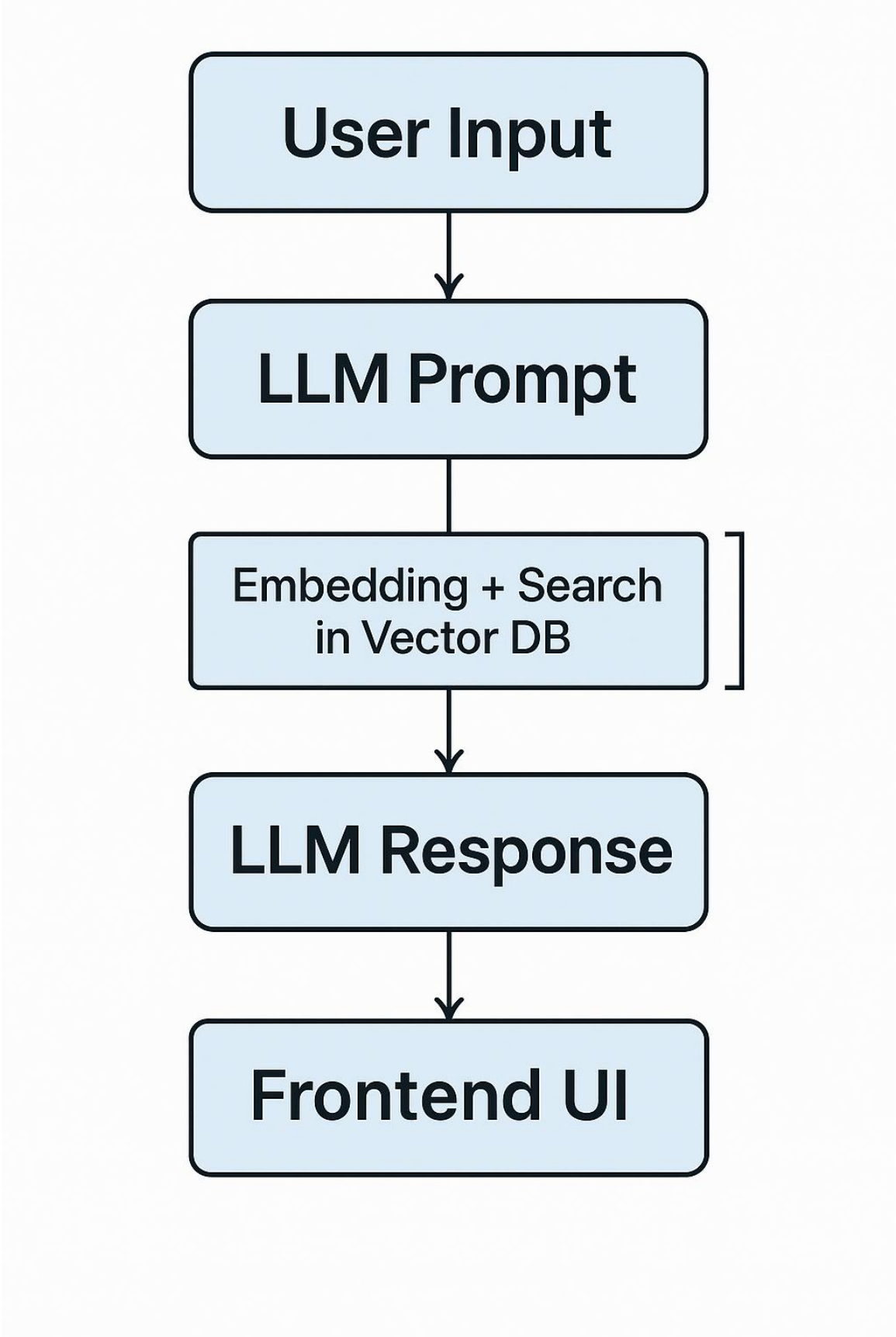
Modern AI tools like **LLMs** (Large Language Models) and **embeddings** let founders integrate powerful capabilities **quickly and safely**, even at the MVP stage.

This section is a cheat sheet to help you understand what's possible and how to make AI a real part of your product, not just a buzzword.

AI Terms You'll Actually Use

Concept	What It Means (Simply)	Used For
LLM	A general-purpose AI model that can read, write, and chat	Chatbots, summarisation, content generation
Embedding	A numeric "fingerprint" of text used to compare meaning	Search, recommendations, semantic filtering
Prompt	A structured instruction sent to the AI	Tells the model what role to play or what to do
Vector DB	A database that stores embeddings and retrieves similar items	Memory, matching, semantic search
RAG	Retrieval-Augmented Generation: blend of AI + search	AI that answers using your company's data

Basic AI-Enhanced Architecture (for MVPs)



Useful AI Tools (You Can Plug In Today)

Task	Tool / API	Why It Works for Startups
Chat / Q&A	OpenAI, Claude, Mistral via OpenRouter	Plug-and-play LLMs for chatbots and forms
Embeddings	OpenAI, Cohere, Bedrock	Turn text into vectors for smarter search
Vector DB	Qdrant, Weaviate, Pinecone	Store and search embeddings efficiently
Prompt Chaining	LangChain, LlamaIndex	Let your prompts evolve across steps or logic
Voice Integration	Vapi.ai, ElevenLabs, Whisper	Add speech-to-text or text-to-speech
Image/Doc Parsing	Azure Form Recognizer, Claude	AI reading files, PDFs, scanned documents

Practical AI Use Cases — No Data Team Needed

Use Case	Description	Tech Stack Example
Semantic Search for Docs	Let users "ask questions" across your docs	Embeddings + Qdrant + RAG
Conversational Chatbot	AI assistant answers FAQs, guides users	OpenRouter + prompt templates
AI-Powered Form Filler	Pre-fills or improves form answers	LLM + user input cleanup
Speech-to-Text Interaction	Users speak, AI responds and extracts meaning	Vapi.ai + Whisper + LLM
Smart Summariser	AI condenses tickets, messages, reports	Claude or GPT-4 + prompt engineering

Final Advice:

*Start small. AI shouldn't be a feature — it should be **a layer that amplifies your product's core value.***

My story

AI has always fascinated me, but I'm also a pragmatist. I've seen too many teams treat AI as a feature or, worse, as a marketing bullet, rather than using it to actually improve the user experience.

In the last year, I've helped several startups **integrate LLMs into real products** — not for novelty, but for real value. These teams had no data scientists, in-house ML infrastructure, and MVP budgets. Still, by using **OpenRouter for chat**, **Amazon Bedrock for embeddings**, **Qdrant as a vector store**, and **FastAPI** for orchestration, we were able to build systems that could search internal knowledge, answer user queries, and even hold lightweight conversations.

In one case, the founder manually answered 20+ pre-sales questions a day. We replaced that with an AI-powered assistant trained on their existing documents. It wasn't perfect, but it cut their workload in half and helped close leads faster.

Another startup wanted users to book meetings by speaking rather than typing. With **Vapi.ai** and **Whisper**, we created a voice-based experience that felt intuitive and natural—something we couldn't have affordably build just a year ago.

More broadly, I believe AI will play a **transformational role in startup knowledge management** — a theme I explored in my **MBA** and ongoing **MSc in AI**. Most early-stage companies lack documentation discipline, and internal knowledge lives in scattered Slack threads, Notion pages, and people's heads. LLMS allow us to build lightweight, context-aware assistants that can answer questions, explain decisions, and surface forgotten insights, without requiring a full-time knowledge manager.

My biggest lesson? Start small and stay grounded. AI is a tool, not a magic wand. Use it to reduce friction, not add new complexity. If your product wasn't useful before AI, it won't be after either.

The Rise of Vibe Coding and AI-Native IDEs

Tools like **Cursor**, **Codeium**, and even GitHub Copilot have changed how we write software. Instead of staring at a blank file, developers can now prompt their way into a working implementation — often faster, and sometimes even more elegant.

This "**vibe coding**" approach — where the AI fills in the boilerplate and scaffolds based on high-level intent — is a game changer for rapid prototyping. You can write less and ship more, especially when building common flows like user auth, form validation, or API integrations.

But here's the caveat:

AI accelerates code writing, not software understanding.

That means you still need a **skilled developer with full context**:

- Someone who can read the AI's output critically
- Spot when the logic is flawed or brittle
- Ensure architecture and patterns remain maintainable
- Debug issues the AI didn't anticipate

In many teams I've worked with recently, I've noticed a shift:

The developer's *main job is debugging AI-written code*, and then optimising, documenting, and integrating it properly.

This isn't a bad thing. It frees time from writing repetitive code and unlocks space for deeper work. I've seen LLMs:

- Generate solid **unit tests** for previously untested code
- Write a clear **README** and endpoint documentation

- Refactor monolith functions into **cleaner components**

So yes — **AI can make you a 10× developer**...
As long as you're still doing the thinking.

Section 4: Data & Security Must-Haves

Secure Enough to Sleep at Night, Without Slowing Down

You don't need an ISO27001 badge to care about security, but you do need **a few good habits** that will save you pain, money, and sleepless nights.

Security is a **non-functional requirement** that investors, partners, and regulators care about, and most of all, your users will expect.

Think of these practices as **"MVP-grade security"** — light enough to move fast, but solid enough to grow on.

MVP Security & Data Checklist

Area	Action to Take	Why It Matters
Encryption	Enable HTTPS + use encryption at rest (e.g. AWS S3, RDS)	Protects data in motion and storage
Secrets Mgmt	Use env vars or tools like Doppler, AWS SSM, or HashiCorp Vault	Avoids hardcoded credentials in repos
Backups	Automate DB & file backups (e.g. AWS RDS snapshots, S3 lifecycle)	Protects against data loss
Access Control	Use least privilege roles in cloud (IAM)	Prevents lateral movement after breach
Audit Logs	Enable basic logging (auth attempts, data changes)	Critical for debugging and incident response
Monitoring	Set up alerts for uptime and error spikes	Early warning before users notice problems
Data Classification	Identify what's sensitive (e.g. PII, payment data)	Helps apply the right level of protection
CI/CD Controls	Protect main/release branches, require reviews	Reduces risk of pushing broken or insecure code
Dependency Scanning	Enable automatic scans in GitHub/GitLab pipelines	Avoids known vulnerabilities in open source libraries
Incident Plan	Keep a one-pager: who does what if something breaks	Saves time and confusion when you're under pressure

My story

I've led teams through **real ISO 27001 audits**, and while the certification process itself can be rigorous, the underlying mindset is something every startup can (and should) adopt early — *without the bureaucracy*.

When I first introduced security practices in a small, fast-moving team, I quickly learned that **the goal isn't paperwork — it's trust**. Trust in your systems, your teammates, and your ability to recover when things go wrong.

We didn't start with dozens of policies or a formal ISMS. Instead, we focused on intent:

- Every decision had **a documented rationale**, even if it was just a short Markdown file
- We made it clear **who owned what** — from infrastructure to access keys
- We automated what we could: backups, logs, 2FA enforcement, and secret rotation

That mindset paid off more than once. In one instance, a third-party tool we used accidentally exposed credentials during a support ticket exchange. Because we had secret rotation set up and access ownership defined, we detected it, responded within minutes, and closed the risk — without any real damage.

But I've also seen preventable mistakes:

- Teams using **the same passwords** across environments
- **S3 buckets left public** "just temporarily"
- GitHub accounts without **2FA**
- Credentials sent over Slack, or **never revoked** when someone left the company

You don't need a full-time CISO to avoid these. You just need **clarity, automation, and a bit of discipline**.

My advice? Start small. Don't aim for certification — aim for **resilience**.
Write down why you did what you did. Know who's responsible. And assume that someday, something will break — so set yourself up to recover fast.

Section 5: Funding Round Tech Prep

How to Look Investable Without Looking Overengineered

Investors may not read your code, but they will assess how seriously you treat your **architecture, scalability, and risk posture**.

Your technical narrative should prove you're not just building fast: you're building **responsibly**, with foresight and structure.

Here's what to prepare when tech due diligence comes into play.

What Investors (and Their CTO Friends) Look For

Area	What They're Thinking	How You Prove It
Architecture	Will this scale if we grow 10×?	Clear stack diagram, infra notes
Security	Can this handle customer data safely?	Checklist from Section 4, clear responsibilities
Team & Process	Is the team able to deliver consistently?	CI/CD, planning cadence, sprint rhythm
Documentation	Could another team understand and maintain this?	Readme, API docs, clean code structure
Tech Debt	Are we investing in a ticking time bomb?	Known issues list + plan to address them
Operational Risk	What happens if someone leaves?	Clear ownership, onboarding, and offboarding steps

Tip: Investors aren't expecting ISO compliance — they're looking for **signals** that your team is mature, coachable, and aware of its risks.

Create a Simple Technical Dataroom (Before You Need One)

Include documents like:

- **1-page System Architecture Diagram**
- **Tech Stack Summary** (with reasoning)
- **Deployment Process** (CI/CD overview)
- **Access Control Policy** (who can touch what)
- **Incident Response Summary** (even a basic one)
- **Dependencies & Known Limitations**
- **Security Measures** (based on your checklist)

Pitch-Ready Tech Diagrams: Keep It Simple

You don't need a 30-page slide deck — just **1–2 clean visuals** that show you're intentional.

What to Include:

- **Frontend–Backend–DB layout**
- **Infra boundaries (e.g., AWS, Vercel, S3, RDS)**
- **Data flows (user input → storage → analytics)**
- **Any external services (e.g., Auth0, Stripe)**

Tools like **Whimsical**, **Lucidchart**, or even **draw.io** work perfectly.

Final Advice:

You don't need to look enterprise-ready — just **investor-ready**.
Show you're building a product with legs, not a prototype with buzzwords.

My Story

Over the years, I've worked with founders preparing for investor meetings where the **tech stack wasn't the main focus, but the tech mindset was**.

At MOYD v1, we didn't start with a full enterprise-grade setup. But when investors came in — especially technical ones — they asked thoughtful questions:

- *What happens if this gets popular overnight?*
- *Can a new engineer understand what you've built?*
- *Who owns what if one of your leads leaves tomorrow?*

We had the answers: not because we were enterprise-ready, but because we'd been **intentional**.

- We had a **clear system architecture diagram**
- A simple document that explained **why we picked our stack**
- Basic CI/CD pipelines and a precise **team rhythm**
- A lightweight **onboarding/offboarding process**

None of it was designed for compliance: it was designed for **clarity**.

In another case, I helped a startup founder assemble a **technical dataroom** after an investor showed sudden interest. We put together diagrams, a deployment workflow, a short note on third-party dependencies, and a bullet-point summary of known tech debt. It didn't look flashy, but it told a coherent story. Two weeks later, that founder got a term sheet.

I’ve learned that **investors don’t expect perfection — they expect foresight**.
They want to see that you’re not building something held together by duct tape and wishful thinking.

If you can show them that you understand your system’s **risks, constraints, and direction**, that’s usually enough to earn their trust and capital.

Closing Thoughts

Startups don't fail because they chose the wrong JavaScript framework.
They fail because they built the wrong thing — too slow, too complex, or without learning.

This toolkit isn't about following rules.
It's about avoiding the traps I've seen (and sometimes stepped into) over two decades of building, scaling, and mentoring technical teams.

If you walk away with one idea, let it be this:
You don't need perfect code. You need fast feedback, stable foundations, and a way to scale when it matters.

You've already done the hard part — starting.
Now build something real.

Want Feedback?

If you want a quick sanity check, a second opinion on your stack, or help shaping your tech story before pitching, just ask.

→ **hello@moyd.co.uk**
→ www.moyd.co.uk