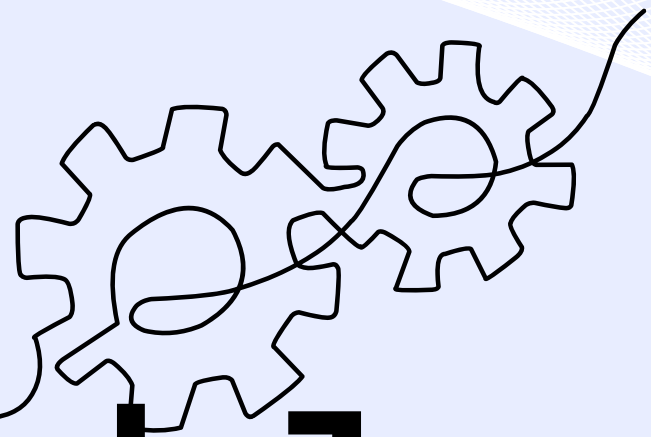

# Tech Jargon *Part 4*

for

Product Managers

## When Old Choices Slow You Down

Tech Debt | Legacy | Refactoring | Migration

*From an Engineer-Turned-PM*

## The Technical Debt & Legacy Systems Mental Mode

A way of seeing your product through the constraints and costs of past technical choices—what they make fast, what they make risky, and what it really takes to change direction.
It helps PMs treat tech debt, legacy systems, and migrations as roadmap-level decisions, not just "engineering cleanup," so you can trade off speed vs sustainability with eyes open.

When you understand this mental model, you can:

- Recognize when "just a small change" is actually tangled in legacy dependencies, and flag that risk early.
- Frame refactors, replatforming, and migrations as clear investments with outcomes (stability, speed, lower cost), not vague "tech debt paydown."
- Work with engineers on deprecation, backward compatibility, and sunsetting plans that minimize customer disruption while steadily retiring old systems.

This guide covers the 10 essential terms that unlock this way of thinking.

# Tech debt

## Accumulated shortcuts that slow future work

Tech debt is the pile of past shortcuts in code, architecture, or process that make new changes slower, riskier, or more expensive.

Think of it as taking construction shortcuts when building a house—skipping insulation and proper wiring saves time early, but you pay for it later with drafts, outages, and costly repairs.

*We can ship this faster now, but we need to be explicit about the tech debt we're taking on and when we'll pay it down.*

**For PMs, this means** you must see debt as a deliberate tradeoff, not a moral failing: know where it is, how it shows up in delivery, and when it's worth paying down to unlock future roadmap items.

# Refactoring

## Improving code without changing behavior

Refactoring is the process of restructuring existing code so it’s cleaner, clearer, and easier to change—without altering what the product does.

Think of it as reorganizing a cluttered closet: you still own the same clothes, but now you can actually find what you need quickly.

*If we refactor this module first, the next few features in this area will be faster and less risky to ship.*

**For PMs, this means** you back refactoring when it reduces bugs, speeds up future feature work, and lowers risk, even if users don’t see an immediate shiny change.

# Legacy system

## Old, hard-to-change system still in use

A legacy system is an older, business-critical system that’s deeply embedded in processes and data, but is difficult and risky to change.

Think of it as an old but vital bridge into a city: everyone relies on it daily, even though it wasn’t built for today’s traffic and is hard to rebuild without causing chaos.

*This feature touches the legacy billing system, so we should expect extra complexity and plan more time.*

**For PMs, this means** you respect the realities of legacy: feature ideas touching it may be slow, costly, or require careful sequencing with migrations and risk mitigation.

# System migration

## Moving from one system to another

A system migration is moving data, workflows, or users from one system or tech stack to another while keeping the business running.

Think of it as moving apartments without pausing your life: your stuff has to move, utilities must keep working, and you can’t lose anything important in the process.

*Our main deliverable this quarter isn’t new features—it’s migrating customers to the new system without disrupting their workflows.*

**For PMs, this means** treating migrations as full projects with cutover plans, data validation, communication, and rollback options—not “just an engineering task.

# Replatforming

## Moving to a new platform with minimal behavior change

Replatforming means taking an existing product and moving it to a new platform (cloud, database, framework) while keeping user-visible behavior mostly the same.

Think of it as renovating the foundation of a building while keeping the same floor plan so residents can continue living there.

*During replatforming, users shouldn't notice big changes; success is better performance and fewer incidents, not a new UI.*

**For PMs, this means** you still gather requirements and manage risk, but your success metrics focus on reliability, performance, and cost—not new UI.

# Strangler pattern

## Gradually replacing a legacy system

The strangler pattern is a way to replace a legacy system piece by piece, routing more traffic to the new system until the old one can be safely turned off.\

Think of it as building a new bridge beside the old one, gradually diverting traffic, and only demolishing the old bridge once the new one handles everything.

*Let's use a strangler pattern: start with one workflow in the new system, then gradually move the rest once we see it working well.*

**For PMs, this means** you plan incremental milestones instead of one big "rip and replace," reducing risk and creating visible progress along the way.

# Backward Compatibility

## New changes still work with old clients

Backward compatibility means new versions of your system still work with existing clients, integrations, or data formats without breaking them.

Think of it as redesigning a plug socket so all the old plugs still work, even though you've upgraded the wiring behind the wal.

*We need this new API version to stay backward compatible so existing integrations don't break overnight.*

**For PMs, this means** you consider who will break if you change an API or data model, and when it's worth maintaining compatibility versus forcing upgrades.

# Deprecation

## Announcing a feature is on its way out

Deprecation is the formal process of marking a feature or API as “on its way out,” communicating timelines and alternatives before actually removing it.

Think of it as putting up “this road will close in 6 months” signs and providing a detour, instead of suddenly blocking the road one morning.

*Let’s announce deprecation now with a clear timeline so customers have time to migrate before we remove this endpoint.*

**For PMs, this means** you own the story: why it’s being deprecated, what customers should do instead, and how long they have to adapt.

# Spike / prototype

## Time-boxed exploration to reduce uncertainty

A spike or prototype is a short, time-boxed experiment engineers run to explore a technical approach, risk, or unknown before committing to a full solution.

Think of it as testing a small patch of paint on the wall before you repaint the entire house.

*We should run a one-week spike to see if this approach is feasible before we commit it to the roadmap.*

**For PMs, this means** you use spikes to de-risk estimates and designs in high-uncertainty areas instead of pretending you have precise answers.

# Sunsetting

## Retiring a feature or system

Sunsetting is the planned end-of-life process for a feature or system, including migrations, communication, and final shutdown.

Think of it as closing a long-running shop with notice, clearance sales, and clear directions to the new location, not just locking the doors overnight.

*If we sunset this feature properly, we'll cut maintenance costs without surprising or upsetting our best customers.*

**For PMs, this means** you lead the change: decide who's impacted, how to help them transition, and how to measure success beyond "we turned it off."


# Up Next

A plain-English guide to how AI features really work—from models and training data to inference, confidence scores, guardrails, bias, and human-in-the-loop review.