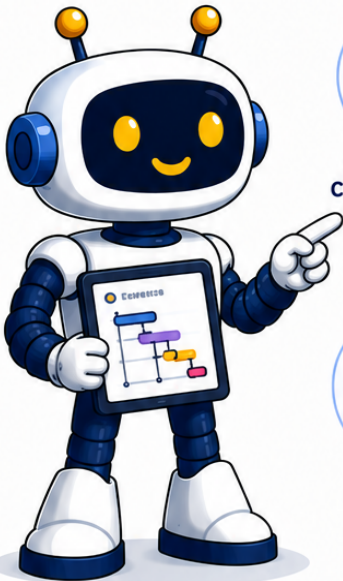
URADICAL

POCKET REFERENCE

PRODUCT MANAGEMENT



Alan Bradley

Product Management Pocket Reference

Owning the product function end to end

Alan Bradley

uradical.io

Product Management Pocket Reference

Alan Bradley

© 2026 uRadical

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

You are free to share and adapt this work for non-commercial purposes, as long as you give appropriate credit and distribute your contributions under the same license.

https://creativecommons.org/licenses/by-nc-sa/4.0/

Table of Contents

1. Preface
2. Discovery
3. Prioritisation
4. Roadmaps
5. Defining Work
6. Metrics
7. Stakeholder Communication
8. The Technical Lead as Product Function
9. Quick Reference

1. Preface

Somewhere between the whiteboard and the user, someone decided engineers could not be trusted with the whole picture. So they hired a middleman.

The dedicated product manager role exists because organisations stopped believing that the people building software were capable of understanding why it should exist. That trust deficit got a job title, a salary band, and a seat in every meeting. The feedback loop that should run directly from user to builder got a node inserted into it — one that filtered, translated, and inevitably distorted.

This is not a book about working with product managers. It is a book about not needing them.

The function of product management is real and necessary. Discovery, prioritisation, roadmaps, metrics — none of this disappears when you remove the role. What disappears is the latency, the information loss, and the process theatre that accumulates around dedicated headcount whose job security depends on being the bottleneck.

When the function lives in the team, engineers build intuition about why they are building things. Feedback reaches the people who can act on it. Prioritisation is made by people who understand the cost of what is being asked.

Good software teams have always done this. They just were not given a framework for it.

This reference gives you that framework. The rest is ownership.

2. Discovery

Discovery is not a phase. It is not a sprint. It is what happens when engineers stay connected to the people they are building for — and what stops happening the moment someone else is paid to do that for them.

Discovery is the practice of understanding the problem before committing to a solution. Teams that skip it do not skip the uncertainty — they just encounter it later, when it costs more to fix.

Talking to Users Directly

Talk to users yourself. Not a report about users. Not a summary of support tickets. A direct conversation, without a translator between you and what they actually said.

Thirty minutes with a real user is worth more than a week of internal debate about what users probably want.

How to run a user conversation:

- Ask about behaviour, not preferences. "Walk me through the last time you did X" beats "What would you like to see?"
- Do not present solutions. Your job is to understand the problem.
- Listen for workarounds. A workaround is a solved problem you did not know about.
- Take verbatim notes. "It's confusing" and "I don't trust it" are different problems.
- Ask why at least twice. The first answer is rarely the real one.

Questions that work:

- What were you trying to do when you ran into this?
- What did you do instead?

- How often does this happen?
- What would have to be true for this to not be a problem?

Questions that do not work:

- Would you use a feature that...?
- On a scale of one to ten, how important is...?
- What features do you want?

Users are experts on their problems. They are not product designers. Do not ask them to design.

Jobs to Be Done

Users do not want features. They have a job to do and they hire software to help them do it. Jobs to Be Done (JTBD) is the framework for making that concrete.

The job statement:

```
When [situation], I want to [motivation], so I can [outcome].
```

Example:

```
When I am reviewing an infrastructure change late at night,
I want to verify the nftables rules haven't drifted from
policy,
so I can approve the change without reading the full ruleset.
```

Features that do not map to a job statement are speculation.

Job types:

Type	Description	Example
Functional	The practical task	Validate config before deployment

Emotional	How the user wants to feel	Confident approving a change
Social	How they want to be perceived	Seen as thorough by the team

Most products address only the functional job. The emotional and social jobs are where differentiation lives.

Extracting Signal from Noise

Signal:

- Repeated friction at the same point in a workflow
- Workarounds users invented themselves
- Complaints about frequency and reliability, not aesthetics
- Requests framed in terms of the user's job, not your product

Noise:

- Feature requests framed as solutions ("I want a button that...")
- Requests from a single user in an unusual situation
- Positive feedback with no specific outcome attached

The test: Can you trace the feedback to a specific user, situation, and outcome? If not, it is noise until corroborated.

The Five-User Rule

Five users is enough to surface the majority of significant problems. After five conversations on the same topic, new users repeat problems you have already heard.

When three separate users describe the same friction unprompted, you have found something worth acting on.

This is not statistical significance. For quantitative claims, you need usage data. Five users is a threshold for qualitative signal, not a substitute for instrumentation.

Writing Up What You Learned

Discovery that stays in your head is memory. Memory decays and cannot be challenged by the rest of the team.

Minimal discovery note:

```
Date:
Users spoken to: (role, not name)
Problem observed:
Evidence: (direct quotes, observed behaviour)
Frequency signal: (one user / multiple users / all users)
Related to: (existing issue, roadmap item, or new finding)
Proposed next step:
```

Keep discovery notes next to the backlog. If they live in a document no one reads, they are not part of the process — they are archaeology waiting to happen.

3. Prioritisation

Prioritisation is an engineering decision. It requires understanding the cost of what is being built, the value it delivers, and the opportunity cost of not building something else. That is not a separate discipline. It is technical judgment applied to product.

Without a deliberate process, the default is whoever shouted loudest most recently. This is a product strategy. Just not a good one.

RICE Scoring

RICE produces a single number for ranking work items across four factors.

Formula:

```
RICE Score = (Reach × Impact × Confidence) / Effort
```

Factors:

Factor	Definition	Unit
Reach	Users affected per time period	Users per quarter
Impact	Effect per user	0.25 / 0.5 / 1 / 2 / 3
Confidence	Certainty in the estimates	Percentage (10–100%)
Effort	Work across the team	Person-months

Impact scale:

Score	Meaning
3	Massive

2	High
1	Medium
0.5	Low
0.25	Minimal

Example:

```
Feature: Bulk export
Reach:       500 users/quarter
Impact:      2 (high)
Confidence: 80%
Effort:      0.5 person-months

RICE = (500 × 2 × 0.80) / 0.5 = 1600
```

Common mistakes:

- Confidence inflation. Most estimates are 50–70%. Be honest.
- Counting impressions as reach. Count distinct users.
- Underestimating effort. Double your first instinct.

MoSCoW

MoSCoW categorises work by necessity within a fixed scope.

Category	Meaning
Must Have	Non-negotiable. Without this, the release fails.
Should Have	Important but not critical. Include if possible.
Could Have	Nice to have. Cut first under pressure.
Won't Have	Explicitly out of scope for this release.

The Won't Have category is as important as the rest. It documents what was decided against and why. Without it, cut items re-enter scope through the side door.

Must Haves should represent no more than 60% of available effort. If everything is a Must Have, nothing is.

Value vs Effort Matrix

Fast triage when scoring models are too slow for the decision at hand.

```
High value .   Quick Wins     .   Major Projects
           .                  .
           ..................................
           .    Fill-ins      .   Avoid / Defer
Low value  .                  .
           ..................................
                 Low effort         High effort
```

Quadrant	Action
High value, low effort	Do first
High value, high effort	Plan and resource properly
Low value, low effort	Fill spare capacity only
Low value, high effort	Do not do

Opportunity Scoring

Opportunity scoring ranks problems by the gap between their importance to users and satisfaction with current solutions. It answers the question that feature requests cannot: where does unmet need actually live?

Formula:

```
Opportunity Score = Importance + max(Importance .
Satisfaction, 0)
```

Where importance and satisfaction are scored 1–10 from user research.

Example:

```
Importance: 8/10   Satisfaction: 3/10
Opportunity = 8 + max(8 . 3, 0) = 13
```

Score	Signal
15+	Strong unmet need. High priority.
10–14	Significant opportunity.
5–9	Moderate. Assess against other priorities.
Under 5	Overserved or low importance. Deprioritise.

Choosing the Right Framework

Situation	Framework
Large backlog, mixed item types	RICE
Release scoping, fixed deadline	MoSCoW
Early triage, limited data	Value vs effort
Problem-level prioritisation from research	Opportunity scoring

Pick one per release cycle. Switching frameworks mid-cycle introduces inconsistency and invites gaming.

Saying No with Data

No is a complete sentence. Data makes it a professional one.

When declining a request, attach a number. A RICE score that ranks the item 47th in the backlog is cleaner than a subjective judgement. A Won't Have with a written rationale is a decision. Without one, it is a dismissal — and dismissals get re-raised.

Template:

```
We reviewed [request] against current priorities.

RICE score: [X] — ranked [N] of [total]
Reason for deferral: [one sentence]
Earliest review: [timeframe or milestone]
Condition for re-evaluation: [what would change this]
```

4. Roadmaps

A roadmap that commits to specific features on specific dates is a project plan with better typography. It will be wrong within weeks and defended for months. Build a better instrument.

A roadmap is a communication tool. It communicates direction and intent. The moment it is treated as a contract, it starts generating two outputs: the work, and the management of the gap between the work and the promise.

Now / Next / Later

The simplest format that works. Three columns. No dates.

```
Now                 . Next                 . Later
.........................................................
What the team       . What the team is     . Direction we
is working on       . working toward       . expect to move in
right now           . after Now            . beyond that
```

- **Now** is specific. Named, scoped, in progress.
- **Next** is directional. Named but not fully scoped.
- **Later** is intentionally vague. Themes and problems, not features. No dates.

Later is not a dumping ground. If something in Later has no real chance of being worked on, remove it. A roadmap padded with aspirational items is a wishlist pretending to be a plan.

Outcome Roadmaps vs Feature Lists

Feature roadmaps list what will be built. Outcome roadmaps describe what will change for users.

Feature roadmap:

```
Q3: Bulk export, SSO integration, audit log filtering
Q4: Mobile app, API v2, custom dashboards
```

Outcome roadmap:

```
Q3: Operators can complete compliance audits without manual
data extraction
Q4: Teams can access the platform from any device without
additional tooling
```

The feature roadmap answers "what are you building." The outcome roadmap answers "what problem are you solving." When a feature slips, the feature roadmap records a failure. When an outcome is achieved via a different feature, the outcome roadmap records a success.

Outcome roadmaps are harder to write because they require you to know why you are building things, not just what. That is a forcing function, not a flaw.

Use feature roadmaps only for fixed external commitments where the feature is itself the deliverable.

What a Roadmap Is Not

A roadmap is not a contract. Nothing on it is a promise unless explicitly committed as one.

A roadmap is not a backlog. The roadmap describes direction. The backlog describes the work.

A roadmap is not a status report. That is a release note.

A roadmap is not a negotiating position. If items exist on the roadmap to satisfy a stakeholder rather than because the team believes in them, the roadmap is already broken.

Communicating Direction Honestly

Stakeholders want certainty. Roadmaps cannot provide it. Teams that pretend otherwise spend more time defending the pretence than doing the work.

Give confidence levels, not dates. Say which items are firm commitments and which are current intentions. Update proactively — a stakeholder who discovers a change without being told will remember it longer than the change itself. When the reasoning changes, say why. "We moved X out because Y turned out to address twice the users at half the cost" is a legitimate product decision. Own it plainly.

The job is not to tell stakeholders what they want to hear. The job is to give them enough information to make good decisions alongside the team.

5. Defining Work

Ambiguous work does not become clear during implementation. It becomes expensive. Every assumption that goes unstated is a decision made by whoever is coding at 11pm with no one to ask.

A well-defined piece of work can be picked up by any engineer on the team, completed without a briefing meeting, and verified against clear criteria. Most work items are not this. That gap is a product function failure.

Writing Specs That Do Not Rot

A spec is a decision record. It exists to prevent the same conversation from happening three times and to capture the reasoning behind the constraints — not to describe every implementation detail before the first line is written.

Minimal spec format:

```
Title:
Status: Draft / Ready / In Progress / Done

Problem
One or two sentences. What is broken or missing, and for whom.

Context
Current state. What has been tried. What we know.

Proposed solution
What we are building. Enough detail to start,
not enough to prevent good decisions during implementation.

Out of scope
Explicit. This section is not optional.

Acceptance criteria
Numbered list. Verifiable statements.
```

```
Open questions
Anything unresolved that could block or redirect the work.
```

What makes a spec rot:

- Acceptance criteria written as intentions rather than conditions
- Missing out of scope section — the most commonly skipped and most expensive omission
- Open questions left open after work starts
- No owner. A spec without an owner is a document, not an agreement.

Acceptance Criteria That Mean Something

Acceptance criteria are a test, not a wish list. They define when work is done. Vague criteria do not save time — they defer the argument to review, or to production.

Criteria that work:

```
- Given a user with admin role, when they request bulk export,
  a CSV is generated and available for download within 30
seconds
  for datasets up to 10,000 rows.

- When the export fails, the user receives an error message
that
  identifies the reason and offers a next step.

- The audit log records the export request, the requesting
user,
  and the row count.
```

Criteria that do not work:

```
- Export should be fast
- Handle errors gracefully
- Log the action
```

The second set requires interpretation at every stage. Each interpretation is a point of failure.

Use Given / When / Then for user-triggered behaviour. Use plain declarative statements for non-functional requirements.

Scope Boundaries

Every piece of work needs an explicit boundary. The scope boundary is not the description of the work — it is the fence. Everything inside is this item. Everything outside is either a separate item or explicitly deferred.

Scope failures are predictable. "Add bulk export" implies format, trigger, progress feedback, error handling, audit logging, and performance at scale. None of those were said. Each one is a decision that will be made ad hoc, under delivery pressure, by whoever is closest to the keyboard.

If a question arises during implementation that the spec does not answer, update the spec before continuing. A guess is not a decision.

Definition of Done

The definition of done is a team-level checklist applied to every piece of work. It is standing policy, not a per-item negotiation.

Example:

```
[ ] Acceptance criteria verified by someone other than the
author
[ ] Tests written and passing
[ ] No new linter errors introduced
[ ] Documentation updated if public-facing behaviour changed
[ ] Monitoring or alerting updated if relevant
[ ] Change noted in release log
```

A definition of done that gets skipped under pressure is a suggestion. If the team cannot hold the line on it, change the list — do not ignore it. An ignored checklist is worse than no checklist because it creates the

appearance of rigour without the substance.

6. Metrics

Output metrics measure how busy the team is. Outcome metrics measure whether it matters. The choice between them is a choice about whether you want to know the truth.

Metrics are how teams know whether the software they are building is working. Without them, product decisions are opinion. With the wrong ones, they are confident opinion — which is harder to correct.

Picking What to Measure

Measure outcomes, not outputs.

Output metric	Outcome metric
Features shipped	Users completing the key workflow
Tickets closed	Error rate in the critical path
Deployment frequency	Revenue per active user
Lines of code	Time to first value for new users

A team can maximise every output metric while making the product worse. This is not a theoretical risk. It is the natural end state of any team measured on activity rather than effect.

The test for a good metric:

- Can it be measured without ambiguity?
- Does it change if and only if the thing you care about changes?
- Can you act on it? A metric that cannot change a decision is decoration.

Track one primary metric and no more than three supporting metrics per product area. More than this and you are measuring everything and acting on nothing.

Leading vs Lagging Indicators

Type	Description	Example
Lagging	Outcomes that have already happened	Monthly revenue, churn rate
Leading	Signals that predict future outcomes	New user activation rate, feature adoption in week one

Lagging indicators tell you whether the product is healthy. Leading indicators tell you why — and give you time to act before the lagging indicators move.

Pairing them:

```
Lagging:  Monthly active users
Leading:  New users completing setup within 24 hours

Lagging:  Churn rate
Leading:  Users who have not logged in for 14 days
```

If the leading indicator moves negatively, you have a window to intervene before it shows up in revenue or retention.

Goodhart's Law in Product

When a measure becomes a target, it ceases to be a good measure.

This is not a fringe risk. It is a structural feature of any system where people are evaluated against a metric they can influence. In product, the gaming is often invisible until the damage is done.

Metric targeted	Gaming behaviour
Daily active users	Forced re-engagement notifications

Net Promoter Score	Survey sent only after successful interactions
Tickets closed	Closed without resolution
Deployment frequency	Trivial, meaningless deploys
Time on site	Dark patterns that trap users

Defences:

- Pair metrics so gaming one costs another.
- Measure user outcomes, not team activity.
- Ask regularly: what would this metric look like if we were optimising for the appearance of success rather than actual success?

Dashboards That Do Not Lie

A dashboard exists to provoke a decision or an investigation. If you look at it and take no action, either everything is healthy — or the dashboard has stopped carrying information and you have mistaken silence for good news.

Show trend, not snapshot. A single number without history is not actionable. Highlight thresholds — the question is not "what is the number" but "is it where it should be." Separate health dashboards from investigation dashboards. If a dashboard requires manual interpretation every time, automate the interpretation. The dashboard should carry the reasoning, not prompt it.

7. Stakeholder Communication

Communication is load-bearing. Teams that communicate badly do not just frustrate stakeholders — they create the conditions for bad decisions to be made above them, and then wonder why leadership keeps interfering.

Communicating direction, progress, and trade-offs is part of running the product function. It is not overhead. It is how the team maintains the autonomy to make good decisions without constant intervention.

Reporting Progress Without Theatre

Status updates that take longer to produce than they save in meetings are bureaucracy with a work-tracking veneer.

Minimal status update:

```
Week of [date]

Shipped
- [item]: [one-line outcome]

In progress
- [item]: [current state, blockers if any]

Next
- [item]: [expected timing if known]

Risks
- [item]: [what could go wrong, what we are doing about it]
```

Ten minutes to write. Ten seconds to read. Replaces a standing status meeting for anyone willing to read it.

The Risks section is the one most teams omit. It is the most important. A risk surfaced early is a problem the team can manage. A risk that surfaces at delivery is a crisis someone else manages — badly, and

loudly.

Frequency: Weekly during active delivery. Fortnightly during planning. Immediately for material changes.

Managing Expectations Under Uncertainty

Most expectation failures are created by the team, not by stakeholders. Stakeholders who are not given information will construct their own estimates. Those estimates will be optimistic. The team will be held to them.

Give ranges, not points. "Two to four weeks depending on what we find in the integration layer" is honest. "Two weeks" is a promise made without the information to keep it.

Flag uncertainty early. The moment an estimate looks wrong, say so. The earlier the signal, the more options exist for responding. Absorbing doubt silently and hoping the timeline recovers is how slips become surprises.

When asked to do more in the same time, make the trade-off explicit:

```
We can hit [date] if we descope [item].
We can include [item] if we move to [later date].
Which matters more?
```

This is not evasion. This is how good decisions get made. The alternative — agreeing to a timeline that cannot be met — is evasion wearing a deadline.

Cutting Scope Without Losing Trust

Scope cuts are product decisions. The failure mode is not the cut — it is silence. Stakeholders who discover a scope cut after the fact remember the discovery. Stakeholders who are told directly, with a reason, move on.

Scope cut announcement:

```
Removing [item] from [release/milestone].

Reason: [one honest sentence]
Impact: [who is affected and how]
Alternative: [what users can do in the meantime, if anything]
Next review: [when this will be reconsidered]
```

Short. Direct. Sent before anyone notices the gap.

8. The Technical Lead as Product Function

The PM role exists because organisations created a gap between engineers and users and then hired someone to stand in it. Close the gap. You do not need the hire.

This is the argument in full: product management is engineering work. It requires technical judgment to prioritise well — because prioritisation means understanding the actual cost of what is being asked. It requires technical credibility to push back on scope — because scope pressure comes from people who do not know what things cost. It requires technical context to interpret what users are asking for versus what they said — because the distance between those two things is where bad products are built.

The claim that this is a separate discipline requiring a separate person with a separate agenda is the story an industry told itself to justify a hiring pattern. The pattern persisted because it spread accountability thin enough that no one was clearly responsible when products failed to serve users.

Engineers who own the product function do not do a second job. They do one job completely.

Owning the Loop

Three functions. One team. One closed loop.

```
Discovery . Prioritisation . Delivery . Discovery
```

Discovery surfaces problems. Prioritisation decides which problems to solve. Delivery builds the solution. Each feeds the next. Cut the loop anywhere and the whole system degrades.

Discovery without prioritisation produces interesting research that never ships. Prioritisation without discovery produces confident decisions made without evidence. Delivery without either produces excellent execution on the wrong things. A team can be busy, well-organised, and hitting every sprint target while building something that does not matter.

The technical lead's accountability is that the loop completes. Others in the team carry parts of it. But someone holds the connection between what users need, what the team decides to build, and what ships.

Sustainable Cadence

The product function is a load management problem. It does not require a separate person — it requires protected time.

Frequency	Activity	Time
Weekly	Discovery write-up from the previous week	30 min
Weekly	Status update	15 min
Fortnightly	Prioritisation review	60 min
Monthly	Roadmap update	60 min
Quarterly	User interviews	3–5 hours total

Twelve to fifteen hours per month. Not zero, not a second job.

User conversations are the first thing to drop under delivery pressure. They are the last thing that should. Discovery debt compounds quietly. A team that has not spoken to users in three months is operating on assumptions. At six months, those assumptions are the product strategy.

When the Team Carries the Function

The strongest version of the product function is not owned by one person. It is owned by the team.

Rotate user interview responsibility. Make prioritisation decisions visible to everyone, not just the lead. Have engineers write their own acceptance criteria and review each other's. Share discovery notes alongside pull requests.

The goal is a team where every engineer has enough product context to make good decisions during implementation — and enough ownership to surface problems before users find them.

This is not a process to install. It is what happens when a team stops treating product judgment as someone else's job. It looks like engineers asking why before asking how. It looks like a conversation about user impact in a code review. It looks like a team that ships things users actually needed.

That is the function. Own it.

9. Quick Reference

RICE Scoring

```
RICE = (Reach × Impact × Confidence) / Effort

Reach:      Users affected per quarter
Impact:     0.25 / 0.5 / 1 / 2 / 3
Confidence: 10% – 100%
Effort:     Person-months
```

MoSCoW

Category	Rule
Must Have	Without this, the release fails
Should Have	Important, include if possible
Could Have	Cut first under pressure
Won't Have	Explicitly out of scope

Opportunity Score

```
Opportunity = Importance + max(Importance . Satisfaction, 0)

15+:    Strong unmet need
10–14:  Significant opportunity
5–9:    Moderate opportunity
<5:     Overserved or low importance
```

Jobs to Be Done

```
When [situation],
I want to [motivation],
so I can [outcome].
```

Now / Next / Later

Now	Next	Later
Named, scoped, in progress	Named, directional	Themes only, no dates

Spec Format

```
Title:
Status: Draft / Ready / In Progress / Done
Problem:
Context:
Proposed solution:
Out of scope:
Acceptance criteria:
Open questions:
```

Acceptance Criteria

```
Given [precondition],
when [action],
then [verifiable outcome].
```

Status Update

```
Shipped:
In progress:
Next:
Risks:
```

Leading vs Lagging Pairs

Lagging	Leading
Monthly active users	New user activation within 24h
Churn rate	Users inactive 14+ days
Revenue per user	Feature adoption in week one
Support ticket volume	Error rate in critical path

Framework Selection

Situation	Use
Large mixed backlog	RICE
Release scoping	MoSCoW
Early triage	Value vs effort
Problem prioritisation from research	Opportunity scoring

Scope Cut Announcement

```
Removing: [item] from [release]
Reason: [one sentence]
Impact: [who is affected]
Alternative: [interim option if any]
Next review: [when]
```

Goodhart Check

```
What would this metric look like if we were optimising
for the appearance of success rather than actual success?
```

Product Management Pocket Reference by Alan Bradley uRadical Ltd — uradical.io Published under Creative Commons CC BY-NC-SA 4.0