The SaaS Replacement Honest Assessment Template

A practical framework for companies evaluating whether to replace SaaS tools with custom software — based on Appunite's real experience replacing Recruitee.

Table of Contents

Section 1

Why This Isn’t a Calculator

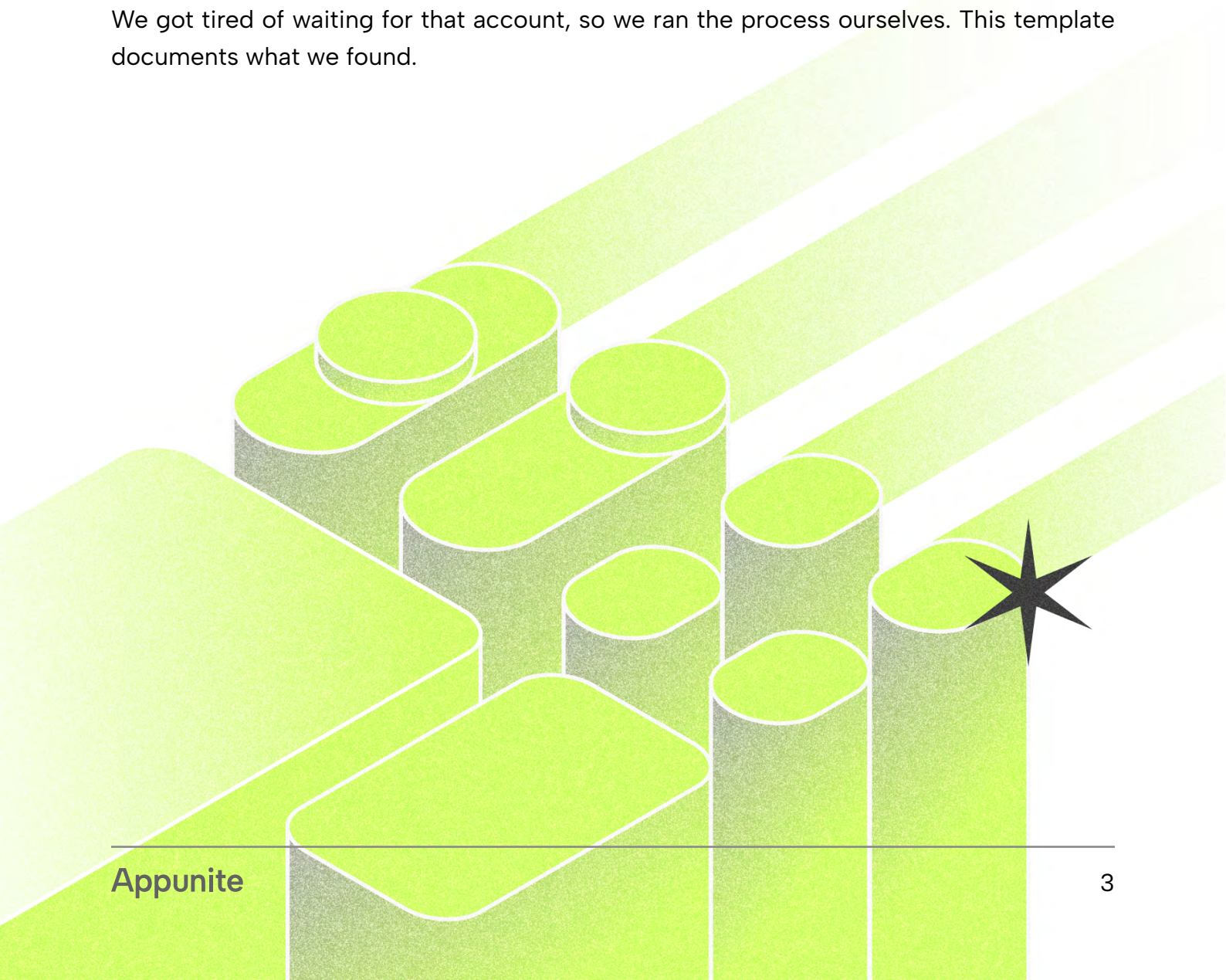
There are two camps in the “SaaS is dead” conversation, and both are exhausting.

The first camp posts LinkedIn threads about how they replaced six tools with one internal build and saved $400k a year. The math is always clean. The timeline is always shorter than expected. Nobody mentions the three engineers who spent eight months on it or the feature they quietly dropped because it was harder than anticipated.

The second camp argues that building is always more expensive than it looks, that maintenance is a hidden iceberg, and that your engineers’ time is better spent on your core product. This is often true. It’s also self-serving.

What’s missing from both camps is an honest account of how the decision actually looks from the inside — before you’ve committed, when you’re still trying to figure out whether the pain is real, whether the numbers hold up, and whether you’re talking yourself into something because you’re frustrated with a vendor or because it genuinely makes sense.

We got tired of waiting for that account, so we ran the process ourselves. This template documents what we found.



The problem with calculators

Search for “build vs buy calculator” and you’ll find no shortage of them. Most work the same way: enter your SaaS subscription cost, your estimated developer salary, your maintenance hours, and the tool spits out a break-even point in months.

These calculators are not wrong. They’re just incomplete in a way that makes them almost useless for the actual decision.

The direct costs — subscription fees, developer time, integration work — are the easy part. They’re visible, they’re defensible, and in most cases they don’t justify building anything. If the decision rested on direct costs alone, the answer is almost always: stay on SaaS.

The hard part is everything else. The reports that calculate wrong so you’re making hiring decisions on bad data. The pipeline stage you can’t edit, so your data is permanently corrupted. The workflow that doesn’t match how your team actually works, so someone exports to a spreadsheet every Monday and manually fixes it. None of that shows up in a calculator because none of it is easy to price.

This is what’s sometimes called opportunity cost — the downstream impact of a tool that doesn’t fit. And here’s the honest problem with it: you can’t prove it. You can estimate that bad hiring data contributed to a failed hire, but you can’t isolate the variable. You can calculate the cost of a failed hire, but you can’t say with certainty the software caused it.

Which creates a trap. Ignore opportunity cost entirely, and the business case for building almost never works. Inflate it, and you’re telling yourself a story that confirms what you already wanted to do.

The right approach is neither. You estimate opportunity cost as rigorously as you can, you assign an attribution percentage you’re willing to defend in a room with your CFO, and then you check whether the case still holds if you’re half wrong. If it only works when your opportunity cost estimates are fully accurate, you don’t have a business case. You have a hypothesis.

This template is built to help you tell the difference.

How we got her

We run Appunite, an Elixir development company. We’ve been arguing for years that AI-assisted engineering is changing the cost equation of custom software — that what was prohibitively expensive to build five years ago may now be viable for internal tools that actually fit how a company works.

At some point it seemed reasonable to test that claim on ourselves. Our ATS was Recrui-

tee. It worked well. But we'd accumulated a list of specific, recurring frustrations — reports that calculated metrics incorrectly, pipeline stages we couldn't edit, GDPR compliance gaps we were patching manually, calendar integrations that didn't cover more than one person. None of it was catastrophic. All of it had a cost.

So we ran the assessment you're about to run. We documented 24 specific pain points, clustered them into problem areas, and tried to put honest numbers on what they were costing us both in direct time and in downstream impact. Then we added up the direct costs and looked at the number. It was 22,524 PLN per year. Our Recruitee subscription was 43,000 PLN. On direct costs alone, building a replacement made no financial sense whatsoever.

Then we added the opportunity cost estimates with explicit attribution assumptions. The number became 150,648 PLN per year. With a two-year subscription cost build budget (86,000 PLN), the ROI flipped to +75%.

One analysis, two completely opposite conclusions, depending entirely on whether you believe the opportunity costs are real.

We decided to proceed. Not because we resolved that question, but because we were honest that we hadn't. The project has additional value for us beyond the direct ROI: learning what AI-assisted engineering can actually deliver, generating content about the process, and stress-testing the thesis we pitch to clients. We're documenting all of it, including the parts where the numbers don't flatter us.

That's the spirit in which this template is written.

How to use this template

This assessment is designed to run across two or three working sessions with a small group. Each section builds on the previous one, and the outputs feed directly into the final decision.

Before you schedule anything, get the right people in the room. This is the step most teams skip, and it's why most assessments end up either too optimistic or shelved without a decision.

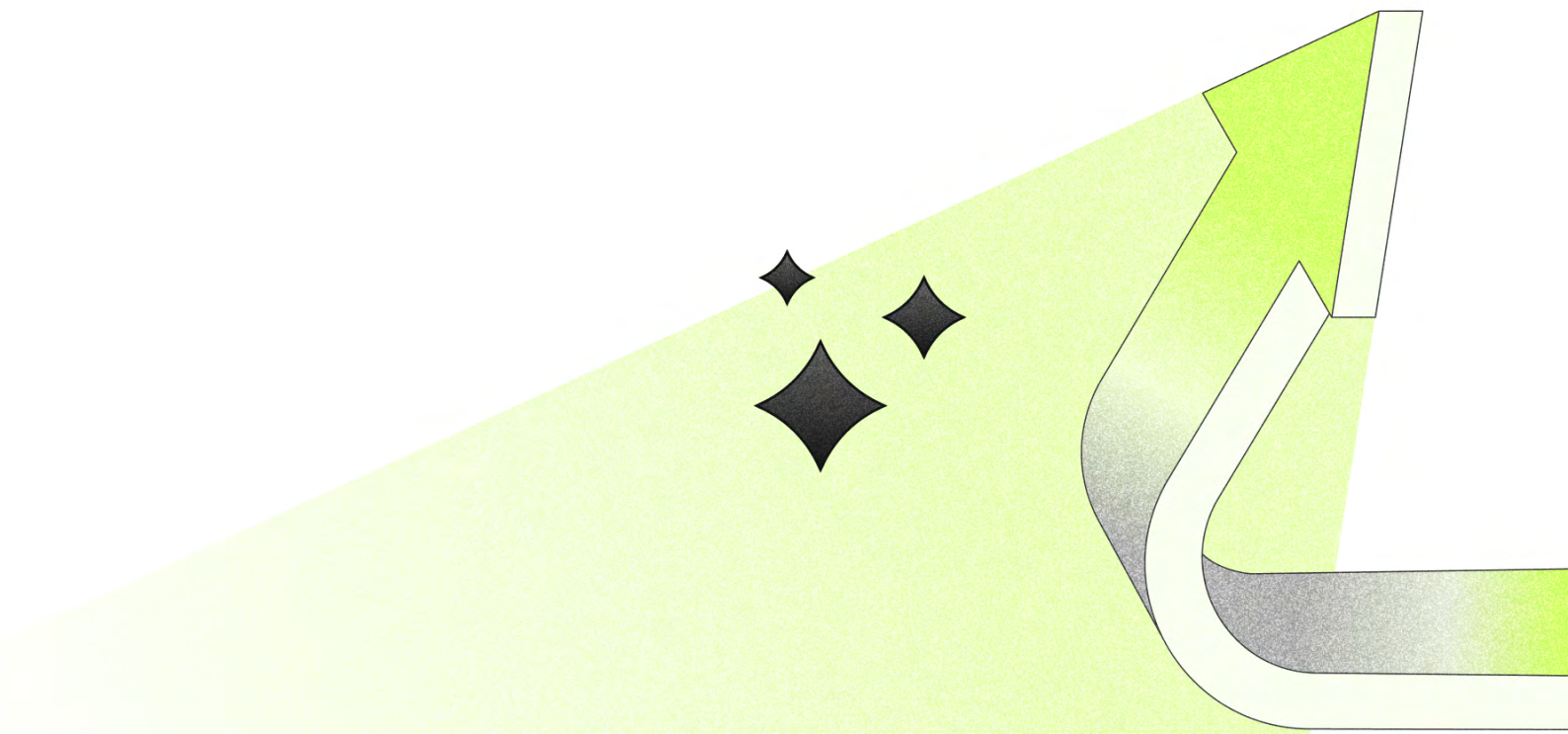
You need three types of people:

- **Someone who uses the tool daily.** The person who runs into its limitations on a Tuesday afternoon and has built workarounds they've stopped noticing. They'll know where the real friction is. They often can't articulate it without prompting, which is why Section 3 includes a structured workshop guide.

- **Someone who can put a number on business impact.** This is usually a finance or operations person, or a founder who owns the P&L. Opportunity cost estimates are debatable by nature. Having someone in the room who will push back on attribution assumptions is not an obstacle — it's the point. An unchallenged opportunity cost estimate is just a wish.
- **Someone with the authority to decide.** Assessments without a decision-maker present tend to produce recommendations that sit in a document for three months. If the person who would approve a build budget isn't involved until the end, you'll spend the last session re-litigating questions that were settled in the first one.

Work through the sections in order. Section 2 (feature usage audit) takes about an hour and usually produces the first genuine surprise — most teams haven't counted what they actually use. Section 3 (pain point discovery) is the most time-intensive and benefits from a dedicated session. Sections 4 and 5 (cost estimation and decision matrix) can be combined if the pain point session was rigorous enough.

By the end, you won't have a calculator output. You'll have a documented position: here's what the tool costs us, here's what we believe but can't prove, here's what building would require, and here's whether we think it's worth it. That's a decision. The calculator was never going to give you one anyway.



Section 2

Feature Usage Audit

Before you estimate costs or evaluate alternatives, you need to answer a more basic question: **how much of the tool you're paying for do you actually use?**

Most teams have never counted. They have a general sense — "we don't use half of it" — but nothing precise. The gap between that intuition and the actual number tends to be larger than expected, and in one direction: teams almost always overestimate how much they use.

According to Pendo's Feature Adoption Report, which analysed usage data across hundreds of software products, 80% of features in the average SaaS product are rarely or never used. Only around 12% of features drive 80% of daily usage volume. These numbers hold across industries and company sizes.

When we ran this exercise on our own Recruitee account, we found we actively used roughly 10% of available features. That's not a criticism of Recruitee — it's how SaaS economics work. Vendors build for a broad market. Your team uses what fits your workflow and ignores the rest. The subscription price reflects the full platform either way.

The purpose of this audit is to make that number concrete for your situation. Once you know what percentage you actually use, you can have an honest conversation about whether you're paying for a platform or for a fraction of one.

Step 01 Build your feature list

Start by generating a complete list of features your SaaS tool offers.

Three ways to get a reliable list:

- **From the vendor's documentation.** Most SaaS tools publish a feature list in their help centre, pricing page, or product changelog. This is the most complete source. Copy every feature into the worksheet below.
- **From the settings or admin panel.** Walk through every menu, tab, and configuration option in your account. This surfaces features that are active in your instance specifically — integrations you've enabled, modules you've purchased, add-ons that come with your plan.
- **From your current team.** Ask two or three daily users to list every action they take in the tool in a typical week. This produces a ground-truth list of what's actually in use — useful as a cross-check once the full audit is done.

Expect to find somewhere between 40 and 150 features depending on the tool. If you're below 20, you're probably working at too high a level (e.g. "reporting" rather than the five distinct report types the tool offers). Be specific.

Step 02 Classify each feature

For every feature on your list, fill in two columns.

Usage frequency — how often does your team actually use this?

Daily

Used by at least one team member on most working days

Weekly

Used regularly but not every day — part of recurring workflows

Monthly

Used for specific periodic tasks (reports, audits, batch processes)

Never

Not used, or used fewer than a handful of times ever

Requirement level — if you were building a replacement, what would you do with this feature?

Required

Without this, the tool doesn't work for your team's core workflow

Nice to have

Useful, but you'd manage without it for at least 6–12 months

Not needed

You wouldn't build it. You're paying for it, but it adds no value

A few notes on common traps:

Don't conflate "used daily" with "required." Some daily habits exist only because the current tool forces them — manual exports, copy-paste workarounds, status updates that exist because there's no automation. If you were building from scratch, you'd solve those differently. Mark them as they currently function, not as you'd ideally want them to work.

Don't mark something as "required" just because it currently exists. Ask: if we built a replacement and this feature wasn't in v1, would we be blocked? If the answer is "inconvenienced but not blocked," it's nice to have.

Feature Audit Worksheet

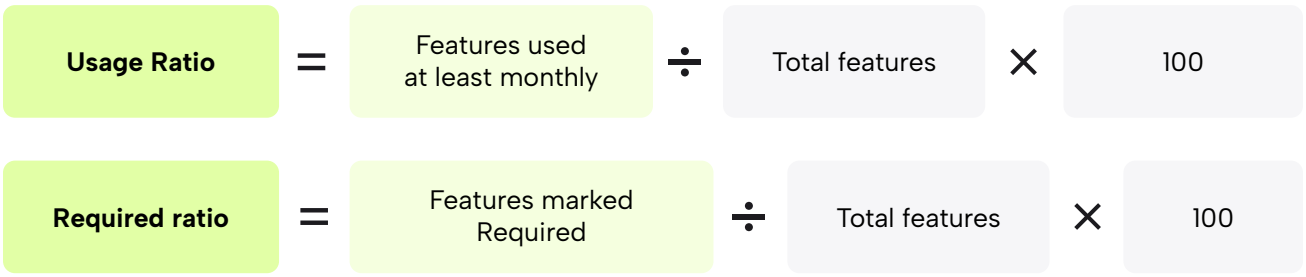
#	Feature	Usage Frequency	Requirement Level	Notes
1		Daily Weekly Monthly Never	Required Nice to have Not needed	
2		Daily Weekly Monthly Never	Required Nice to have Not needed	
3		Daily Weekly Monthly Never	Required Nice to have Not needed	
4		Daily Weekly Monthly Never	Required Nice to have Not needed	
5		Daily Weekly Monthly Never	Required Nice to have Not needed	
6		Daily Weekly Monthly Never	Required Nice to have Not needed	
7		Daily Weekly Monthly Never	Required Nice to have Not needed	

Step 03 Calculate your usage ratio

Once the worksheet is complete, total up the following:

- **Total features listed**
- **Features used at least monthly** (Daily + Weekly + Monthly)
- **Features marked Required**

Then calculate:



The usage ratio tells you how much of the platform you actually engage with. The required ratio tells you the minimum footprint a replacement would need to cover.

If your required ratio is above 20%, either the tool is genuinely well-matched to your needs (which is worth knowing before you consider replacing it), or your team has built its workflow around the tool's structure rather than the other way around. Both are worth examining before you proceed.

If your required ratio is under 15% — which in our experience is common for internal back-office tools — you're in territory where a custom build covering that 15% could plausibly replace the full platform.

What to do with the result

The feature audit doesn't make the decision but it frames the next conversation.

If you find you're using 40% of the platform and 30% is required, the economics of building look different than if you're using 10% and 8% is required. The audit also tends to shift the framing in the room: abstract frustrations about the tool become concrete numbers, and concrete numbers change how teams talk about cost.

Bring the completed worksheet into your pain point discovery session (Section 3). Cross-reference the features marked "Used daily" or "Required" against the pain points your team identifies. Where you find features that are both required and painful — things you can't live without but that don't work as needed — that's where the case for building is usually strongest.

Section 3:

Pain Point Discovery

The feature audit tells you how much of the platform you use. **This section tells you what it's costing you.**

These are different questions. A feature can be used daily and still be a source of genuine pain — it's required, it doesn't work well, and your team has built workarounds they've stopped noticing. The goal of this session is to surface those frictions, describe them precisely, and separate the ones that are genuinely caused by the software from the ones that would follow you into any tool because they're actually process problems.

That last distinction is the hardest and the most important. **The most common way this kind of assessment goes wrong is that teams use it to confirm a decision they've already made** — they blame the tool for everything, the numbers come out in favour of building, and six months later they discover that three of their biggest pain points exist in the new system too because the root cause was never the software.

Run this session honestly, especially the solvability filter at the end.

Workshop setup

Who to invite

Apply the same principle from Section 1: **you need daily users, someone who can speak to business impact, and a decision-maker**. For this specific session, the composition matters slightly differently.

Daily users are the core of this workshop — they hold the detail. Aim for two or three people who use the tool regularly across different parts of the workflow. If one person uses it for sourcing and another for scheduling and another for reporting, you want all three. Pain points tend to be invisible to people who don't encounter them directly.

Limit the group to six people. Above that, the session becomes a meeting instead of a working session, and quieter participants stop contributing.

Do not invite the person who owns the vendor relationship or who made the original buying decision, unless they are also a daily user. They have a natural incentive to defend the status quo and will often reframe pain points as user error or training gaps. You can share the output with them afterwards.

How long

Plan for 90 minutes. The first 20 minutes are collection — participants log their pain points individually before any group discussion, which prevents the first vocal person from setting the frame for everyone else. The next 50 minutes are discussion, clarification, and scoring. The final 20 minutes are for the solvability filter.

If the group is surfacing more pain than expected and energy is high, extend to two hours. Don't rush the scoring — a pain point logged as severity 3 when it should be a 5 will understate the business case in Section 4.

What to prepare

Share the pain point collection template (below) with participants at least 24 hours in advance (we gave ourselves a week) **and ask them to fill in at least five entries before they arrive.** People recall pain more accurately when they're doing the work than when they're trying to remember it in a room. Blank templates filled in during the session produce shallower outputs.

Pain point collection template

For each problem, fill in every field. Partial entries — especially missing workarounds and time estimates — make the cost calculation in Section 4 unreliable.

	What to capture
Description	What specifically happens, or fails to happen? Be concrete. "Reporting doesn't work well" is not a pain point. "The hire count in the dashboard always shows 0%, regardless of actual hires" is.
Frequency	How often does this affect someone on the team? Daily / Weekly / Monthly / Quarterly
Severity	Score 1–5 using the anchors below. Don't score by feel — score by consequence.
Current workaround	What does the team actually do today to work around this? If the answer is "nothing, we just live with it," that's a valid answer — and usually means the consequence is either trivial or quietly absorbed somewhere.
Time per occurrence	How many minutes does the workaround take each time? Estimate conservatively. You will use this number in Section 4.

Severity scale — anchored to business consequence

Score	What to capture
★☆☆☆☆	Annoying but no workaround needed. Work continues normally.
★★☆☆☆	Requires a workaround, but it takes under 5 minutes and is reliable.
★★★☆☆	Workaround exists but is time-consuming or introduces data risk.
★★★★☆	No clean workaround. Either data quality suffers or someone absorbs significant manual work on a recurring basis.
★★★★★	Blocking. The task cannot be completed as intended, compliance is at risk, or the problem causes failures in downstream workflows.

The difference between a 3 and a 4 is whether the workaround leaves a residue. A workaround that takes 10 minutes and is done is a 3. A workaround that introduces corrupted data you'll need to reconcile later, or a manual step that occasionally gets missed and causes a downstream problem, is a 4.

A 5 is reserved for problems where someone is genuinely stuck, not inconvenienced. In our experience, most real-world ATSs have one or two of these. If your list has ten 5s, recalibrate.

Clustering methodology

Once the group has logged individual pain points and discussed them, the next step is grouping them into clusters. The reason to cluster rather than work with individual items is that individual pain points often share a root cause, and the cost and solvability of the cluster is what matters for the decision — not the cost of any single friction.

How to cluster

Group pain points first by the feature area they relate to — the part of the tool where the problem lives. Scheduling, reporting, pipeline management, candidate records, integrations, and so on. This produces natural groupings based on where in the tool the friction occurs.

Then, for each group, write a single sentence that names the cluster by its business consequence. It's where the framing shifts from a complaint about software to a statement about cost.

The sentence structure that works: [Feature area] does not [capability], which prevents us from [business outcome] / results in [business consequence].

For example:

- *Reports are not elastic and calculate with errors, preventing us from seeing accurate time-to-hire and cost-per-hire metrics.*
- *Funnels are not editable after the fact, resulting in data corruption and unreliable pipeline metrics.*
- *It is not possible to design competency matrices, preventing us from systematically screening past candidates for new roles.*

The business consequence in that sentence is what you will price in Section 4. Name it precisely now — vague cluster names produce vague cost estimates.

What a good cluster looks like

A well-formed cluster has three to eight individual pain points underneath it, a single dominant feature area, and a business consequence you can articulate to someone outside the team in one sentence. If a cluster has fifteen pain points under it, it probably contains two clusters. If you can't articulate the business consequence, the cluster isn't ready for Section 4.

In our Recruitee assessment, 24 individual problems clustered into 7 groups. That ratio — roughly 3–4 pain points per cluster — is typical for a tool that's been in use for a year or more.

The solvability filter

Before you take clusters into the cost estimation in Section 4, run each one through three questions. This is the step that determines whether a pain point is a reason to build, a reason to switch, or a reason to look in the mirror.

Question 1 **Is this a software problem or a process problem?**

A software problem is one where the tool genuinely cannot do what the workflow requires — the feature doesn't exist, the data model doesn't support it, the integration isn't available. A process problem is one where the tool could support a better workflow if the team changed how they work.

These are easy to confuse because the symptom looks identical: something isn't working. The test is to ask: if we had unlimited configuration options in the current tool, would this problem still exist? If yes, it's a process problem. Build won't fix it.

If the answer is genuinely unclear, that's a signal to probe further in the workshop before proceeding.

Question 2 **Could a different SaaS solve this?**

If the pain is real and software-caused, the next question is whether another tool on the market already solves it. Switching SaaS tools is almost always cheaper and faster than building. The bar for building should be: no existing product solves this adequately, or the switching cost across all pain points exceeds the build cost.

If a different SaaS would solve two of your seven clusters, factor that into the scope. You're not necessarily replacing the entire tool — you may be patching two specific gaps with integrations or switching one module.

Question 3 **Does solving this require data ownership or custom logic that no SaaS can provide?**

This is the question where building typically wins the argument. If the pain requires the ability to query your own data in ways the vendor doesn't expose, to build workflows that don't exist in any available tool, or to maintain persistent context about candidates or customers that lives outside any vendor's data model — that's where custom software has a structural advantage that switching cannot address.

In our case, the ability to track candidate relationships over time, surface past candidates for new roles based on competency data, and own the full history of every interaction fell into this category. No ATS we evaluated offered this as a native feature, and the ones that approximated it did so through vendor-controlled data structures we couldn't modify.

Solvability filter worksheet

Cluster	Software or process problem?	Could a different SaaS solve this?	Requires data ownership or custom logic?	Verdict
	Software Process Unclear	Yes Partially No	Yes No	Build candidate Switch candidate Fix the process
	Software Process Unclear	Yes Partially No	Yes No	Build candidate Switch candidate Fix the process
	Software Process Unclear	Yes Partially No	Yes No	Build candidate Switch candidate Fix the process
	Software Process Unclear	Yes Partially No	Yes No	Build candidate Switch candidate Fix the process
	Software Process Unclear	Yes Partially No	Yes No	Build candidate Switch candidate Fix the process
	Software Process Unclear	Yes Partially No	Yes No	Build candidate Switch candidate Fix the process

Clusters that come out as “Build candidate” on the solvability filter, with severity 4–5 pain points underneath them, are the core of your business case in Section 4. Clusters that are process problems or switchable to another SaaS don’t belong in the cost justification for building — including them inflates the numbers and weakens the argument if anyone scrutinises it.

Section 4

Cost Estimation Framework

You now have a list of pain point clusters, each with a business consequence and a solvability verdict. This section turns those into numbers.

Before you start calculating costs, establish your build budget ceiling. The rule of thumb we use: **your maximum build budget should not exceed two years of your current SaaS subscription cost, plus 20%**. The logic is straightforward — if you spend that entire budget, the break-even point is two years, assuming the custom software solves 100% of the identified problems and your opportunity cost estimates are accurate. Both assumptions are optimistic, which is why the ceiling exists. Exceed it, and the economics become very difficult to defend.

For example: if your annual SaaS subscription is 43,000 PLN, your budget ceiling is (43,000 × 2) × 1.2 = 103,200 PLN. In practice, you should aim to come in well below that ceiling — the 20% is a margin for the inevitable.

The goal is to **build an honest business case** that holds up when someone in the room pushes back, and one that tells you clearly whether the project makes financial sense or whether you're proceeding on belief.

There are two types of cost to calculate. Direct costs are the time your team loses to workarounds — calculable, defensible, and rarely sufficient on their own to justify building. **Opportunity costs** are the downstream business impact of the tool's limitations — harder to isolate, usually larger, and where most build-vs-buy analyses either inflate the case or avoid the question entirely.

You need both. But you need to keep them visibly separate, because they carry very different levels of confidence.

Step 1 Direct cost calculation

For each pain point cluster that passed the solvability filter in Section 3, calculate the annual cost of the time your team spends working around the problem.

The formula:

$$\textbf{occurrences per month} \times \text{minutes per occurrence} \times \frac{\text{hourly rate}}{60} \times 12 = \textbf{annual direct cost}$$

A note on hourly rate: use fully-loaded cost, not base salary. Fully-loaded cost includes employer taxes, benefits, equipment, and office overhead — typically 30–40% above the gross salary figure. Using base salary understates the real cost of someone's time and makes Column A look smaller than it is. That's a mistake in the direction of optimism, which is the direction this kind of analysis tends to err in by default.

If you don't know the fully-loaded rate, a reasonable working assumption for most knowledge worker roles is to take the monthly gross salary and multiply by 1.35. Divide by 168 (average working hours per month) to get an hourly figure.

Direct cost worksheet:

Cluster	**Occurences per month**	**Annual direct cost**	**Hourly rate** (PLN/EUR)	**Monthly direct cost**	**Annual direct cost**
TOTAL					

Sum the annual direct cost column. This number is your floor — the cost you can defend to anyone without needing to make assumptions. Keep it visible throughout the rest of this section.

Step 2 Opportunity cost estimation

Direct costs rarely tell the full story. The more significant cost of a poorly-fitted tool is usually downstream: decisions made on bad data, processes that slow down because the tool creates bottlenecks, and outcomes that degrade because context is lost.

These costs are real. They're also genuinely difficult to attribute to any single cause, which is why most build-vs-buy analyses either ignore them (making the case look weaker than it is) or include them uncritically (making it look stronger than it deserves).

The framework below is designed to force a middle path: estimate the downstream impact, then force yourself to assign an attribution percentage — the fraction of that downstream impact you're willing to claim is actually caused by the software limitation.

For each cluster with a plausible downstream consequence, fill in the following:

Field	What to capture
Downstream consequence	What business outcome degrades because of this cluster? Be specific: not "worse hiring" but "increased failed hire rate" or "longer time to fill, costing us N weeks of vacancy per role."
Consequence size	What is the financial value of one occurrence of this consequence? Failed hire cost, cost of an unfilled role per week, revenue impact of a delayed process.
Frequency	How often does this consequence occur per month?
Gross opportunity cost	Consequence size × frequency × 12 = annual gross opportunity cost if 100% attributable to the software.
Attribution %	What percentage of this consequence is actually caused by the software limitation, as opposed to other factors (hiring market, team capacity, process quality)?
Attributed opportunity cost	Gross opportunity cost × attribution % = the number you'll carry forward.

The attribution percentage is the uncomfortable part, but it's extremely important. A 50% attribution on a large downstream consequence will swing your total cost significantly — and no formula tells you the right number. You are making a judgment call, and you should document your reasoning for it.

When we ran this on our own analysis, our single largest opportunity cost assumption was that 50% of failed hires were attributable to inaccurate reporting data — specifically, that we couldn't see which pipeline stages were bottlenecks or which sources produced candidates who stayed. That one assumption drove 67% of our total estimated opportunity cost. Changing it from 50% to 0% turned a +75% ROI into a -74% ROI.

That's a swing that should make anyone uncomfortable. Document yours with the same specificity, so the decision-maker in the room is choosing to accept or reject a named assumption, not a black box number.

Step 3 The break-even attribution test

Once you have your gross opportunity costs and your attribution estimates, run this test before proceeding to the summary table. For each opportunity cost line, ask: **at what attribution percentage does this line stop contributing to a positive ROI?**

In practice: take your total project build cost, subtract your total direct cost savings, and calculate how much of the opportunity costs you need to cover the gap. Divide that by your gross opportunity cost to get the minimum attribution percentage required for break-even.

If break-even requires 80% attribution on a consequence you've estimated at 30%, your case depends on an assumption you don't believe. That's not a reason to abandon the project — but it is a reason to be explicit that you're proceeding for reasons beyond the direct financial case.

If break-even is achievable at 30–40% attribution, and your estimate is 50%, you have reasonable margin. The project makes sense even if you're half-wrong.

In our case, the threshold was approximately 50% of estimated opportunity costs being real. We couldn't prove they were. We documented that clearly, noted the additional value the project would generate beyond direct ROI (learning, content, proof of concept for the thesis we pitch to clients), and proceeded with that on the record.

Step 4 The honest split

Bring everything together into a single summary table with two columns.

	Column A **Direct costs only**	Column B **Direct + attributed opportunity costs**
Downstream consequence	Sum of direct costs only	Direct costs + attributed opportunity costs
Project build budget	[Your number]	[Same number]
ROI	(Column A − build budget) ÷ build budget	(Column B − build budget) ÷ build budget
Payback period	Build budget ÷ Column A savings	Build budget ÷ Column B savings
Break-even attribution required	N/A	Minimum % of opportunity costs that must be real

If Column A ROI is positive, your case is strong. You don't need to rely on opportunity costs to justify the decision, which means the decision is defensible under scrutiny.

If Column A ROI is negative and Column B is positive, you are making a bet on your attribution assumptions. That's not automatically the wrong call — but it needs to be framed honestly, and it raises the question of whether you can validate the assumptions before committing the full build budget.

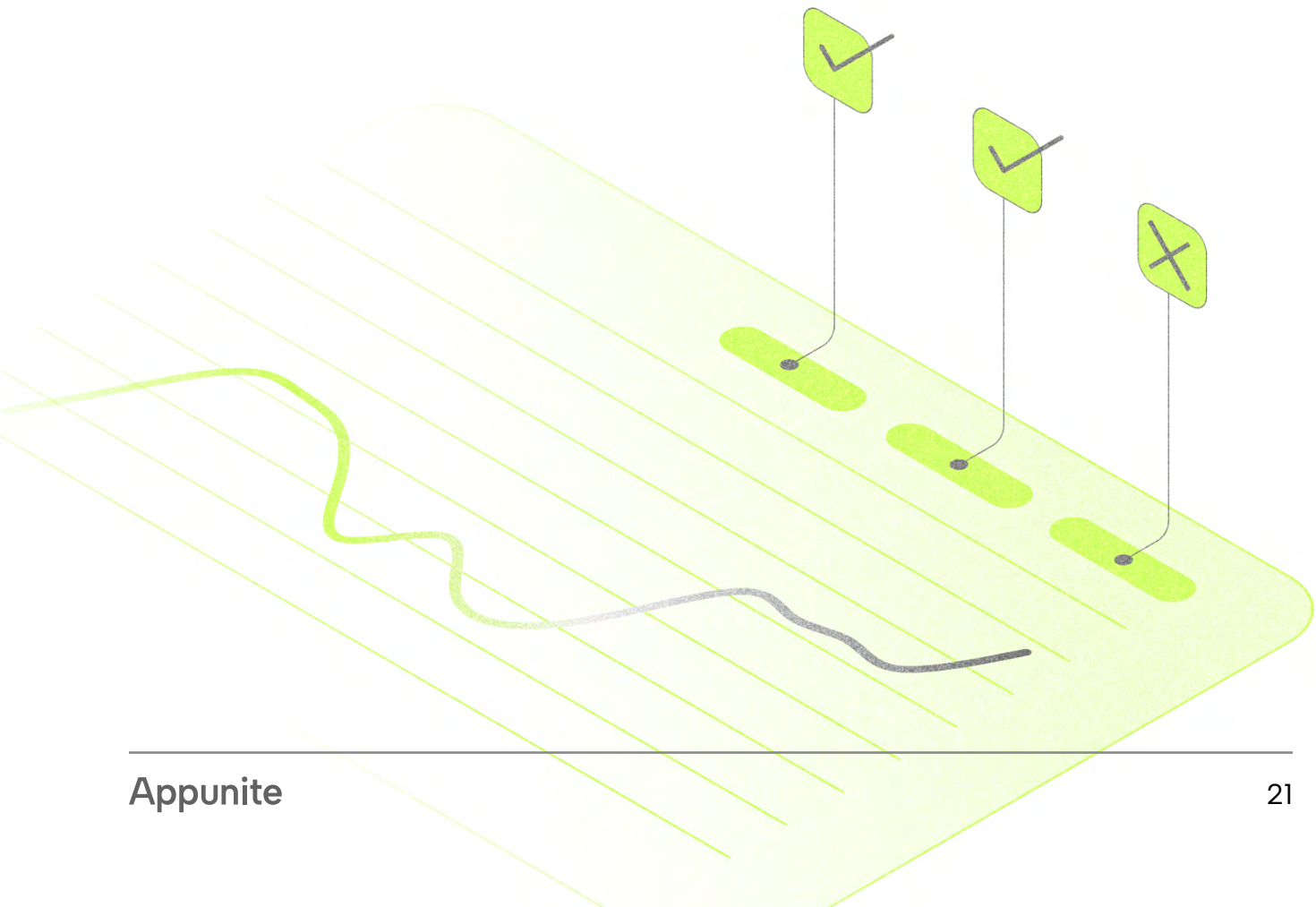
If your case only works in Column B, do this before proceeding:

Pick the two or three attribution assumptions that contribute most to the gap between Column A and Column B. For each one, design a quick validation: not a research project, but a sanity check you can run in a week or two.

For a "bad data > bad decisions > failed hires" assumption: pull your last 12 months of hiring outcomes and look for a pattern. Did failed hires correlate with roles or stages where your reporting data was least reliable? You won't get a clean answer, but you'll get a directional signal.

For a "manual workaround > process delays > slower time-to-hire" assumption: time a single recruitment cycle end-to-end and break it down by stage. How much of the elapsed time is tool-caused friction vs. decision time vs. candidate availability?

These validations won't give you certainty. They'll give you enough to decide whether the attribution assumption is in the right order of magnitude — which is all you need to proceed with appropriate confidence, or to stop and acknowledge the business case isn't there.



Section 5

The Decision Matrix

You now have pain point clusters, solvability verdicts, and two columns of cost estimates. This section is where you make the actual decision.

There are three possible outcomes: build a custom replacement, switch to a different SaaS, or stay with what you have and fix the process. Most assessments are implicitly designed to arrive at one of these before the analysis starts. The purpose of this section is to resist that pull and let the work you've done actually determine the answer.

Work through the decision tree below in order. Don't skip steps because the answer seems obvious — the steps where the answer feels obvious are exactly where the most expensive mistakes get made.

The decision tree

Step 1 **Are your pain points primarily software problems or process problems?**

Go back to the solvability filter from Section 3. Count how many of your clusters came out as “software problem” vs. “process problem” vs. “unclear.”

If the majority are process problems, stop here. Building new software will not fix a process problem — it will give you a new tool in which the same process problems exist. Fix the process first, then reassess the tool. Process work takes weeks. A build takes months and costs significantly more. Do the cheaper experiment first.

If you have a mix — some clusters are software problems, some are process — this is the most common real-world outcome and the one most decision trees ignore. Don’t treat the mix as “mostly software” and proceed; separate the two: commit to fixing the process components regardless of what you decide about the tool, and then evaluate whether the remaining software-caused pain is sufficient to justify building. Carry only the software-caused clusters into Step 2.

If the picture is genuinely unclear on one or two clusters after the solvability filter, flag them as undecided. Don’t assign them to either column to make the numbers work.

Step 2 **Could a different SaaS solve the core pain points?**

For the clusters that are genuine software problems, check whether another tool on the market solves them. This is a step most teams underinvest in because switching feels disruptive. Run it seriously anyway.

A useful shortcut: take your three highest-severity clusters and evaluate two or three alternative tools specifically against those pain points. Not against feature lists — against the specific business consequence you documented in Section 3. A tool can have a “reporting” module and still not solve your reporting problem.

Switching is almost always cheaper than building — faster to implement, lower upfront cost, no maintenance responsibility. **The case for building over switching requires at least one of the following to be true:**

- No available SaaS solves your highest-severity clusters adequately
- The switching cost across all affected workflows is comparable to or exceeds the build cost
- The solution requires capabilities that are structurally unavailable in the SaaS model: custom data logic, persistent context you own, workflows proprietary enough that no vendor has built for them

If a different SaaS solves most of your pain at a switching cost clearly below your build estimate, the decision is straightforward. Switch, revisit in 12 months.

If switching solves some clusters but not others, calculate it as a partial solution: what does the residual pain cost after switching? If the residual pain justifies a targeted build on top of the new SaaS, that's a hybrid path worth considering. If the residual pain is manageable, switching alone may be enough.

Step 3 Do your core pain points require something no SaaS can provide?

This is the question where building earns its place in the conversation.

There are three types of requirements that the SaaS model is structurally unable to meet, regardless of which vendor you choose:

- **Custom business logic.** If your process has rules, exceptions, or workflows specific enough that no general-purpose tool has modeled them, and configuration doesn't get you there, the tool will always be a partial fit. You'll keep building workarounds forever.
- **Data ownership and queryability.** SaaS platforms own your data inside their model. You can export it, but you can't query it, join it, or build on top of it the way you could with a database you control. If your use case requires running custom analytics, surfacing historical context, or building feedback loops between data points the vendor doesn't connect — that requires ownership.
- **Proprietary workflows as a competitive advantage.** If the process the tool supports is something your organisation does differently from the market — and that difference is a source of competitive advantage — using the same tool as everyone else erodes that advantage. Custom software is one of the few ways to encode a proprietary process in a way that doesn't get standardised away.

In our case, all three applied to candidate relationship management: the logic for surfacing past candidates for new roles based on competency history is specific to how we recruit, the data model requires ownership to build correctly, and the ability to maintain genuine long-term relationships with candidates rather than treating each process as isolated is something we believe creates a hiring advantage. No ATS we evaluated was built around this model.

If your core pain points land in at least one of these three categories, custom build is a legitimate option. Move to Step 4.

Step 4 **Does the financial case hold under scrutiny?**

Let's return to the honest split table from Section 4.

	Column A **Direct costs only**	Column B **Direct + attributed opportunity costs**
Downstream consequence	Sum of direct costs only	Direct costs + attributed opportunity costs
Project build budget	[Your number]	[Same number]
ROI	(Column A − build budget) ÷ build budget	(Column B − build budget) ÷ build budget
Payback period	Build budget ÷ Column A savings	Build budget ÷ Column B savings
Break-even attribution required	N/A	Minimum % of opportunity costs that must be real

If your Column A ROI (direct costs only) is positive within two years, the case is strong. You don't need to rely on opportunity cost assumptions that can be challenged. Proceed.

If Column A ROI is negative, calculate the break-even attribution threshold: what percentage of your estimated opportunity costs would need to be real for the project to break even within one year? In our case, that threshold was 50% — we needed at least half of our opportunity cost estimates to be accurate for year-one break-even. We couldn't prove that. We documented it clearly and proceeded because the project had additional value beyond direct ROI. That was a legitimate decision with the uncertainty on the record.

If your break-even threshold is above 70–80% — meaning the project only works if nearly all your opportunity cost assumptions are accurate — that's a fragile case. Either validate the assumptions first (see Section 4), reduce the build scope to lower the investment required, or be explicit that you're proceeding on strategic grounds rather than financial ones. All three are defensible. Proceeding while believing the numbers are solid when they aren't is not.

When not to build

These are the conditions under which building is the wrong answer, regardless of how the decision tree comes out. They are based on what we've seen fail, not on theoretical risk.

- **The process the tool supports is not business-critical.**

  This is the first thing to check, and it cuts in one direction: if the process isn't business-critical, the investment probably isn't worth making. Changing tooling is disruptive — it takes engineering time, migration effort, and people have to rebuild habits. That disruption is justified when the process directly supports revenue, hiring, compliance, or another outcome that materially affects the business. It is not justified for peripheral workflows that affect a small number of people in ways that don't connect to business outcomes. Before committing to a build, ask: **if this process degrades, does the business feel it**? If the honest answer is no, spend your engineering capacity elsewhere.

- **The pain is real but affects fewer than three people.**

  Custom software has a fixed overhead: architecture decisions, deployment, security, future modifications. That overhead doesn't scale down to single-user problems. If one person is frustrated with a tool and two others can work around it, the economics rarely justify building. The threshold is whether the problem is structural enough to affect a workflow, not just an individual preference.

- **You'd be rebuilding commodity functionality.**

  Email. Calendar. Basic data entry forms. User authentication. Document storage. These are solved problems with dozens of mature, cheap implementations. Building them yourself is expensive, time-consuming, and produces infrastructure you'll need to maintain forever. If your core pain points are in commodity territory, the argument for building collapses — not because building is wrong, but because you're building the wrong thing. The value of custom software is in the logic specific to your business, not in reimplementing what already exists.

- **You cannot articulate what custom software would do that no SaaS can.**

  This is the shiny object test. If the answer to "what would this tool do that we can't buy?" is vague — "it would be ours," "it would fit better," "it would be more flexible" — you don't have a build case yet. The articulation doesn't need to be long. It needs to be specific: this tool would do X, which requires Y capability, which no vendor provides because Z. If you can't complete that sentence, do the earlier sections of this template again before proceeding.

Section 6

Scoping Guardrails

If you've worked through the previous sections and the decision is to build, the most likely way this goes wrong is no longer financial. The financial case is documented. The risk now is scope.

Custom software projects fail in a predictable pattern: the initial scope is reasonable, then requirements accumulate, then the build takes twice as long as estimated, then the budget is gone before the thing works well enough to replace the tool it was meant to replace. Every addition seemed justified at the time. The cumulative effect was a project that couldn't deliver what it promised within the constraints that made it viable.

These guardrails exist to prevent that.

The 10% rule

You completed the feature audit in Section 2. Your required ratio — the features marked as genuinely required divided by total features — is your v1 build scope.

In practice, for most internal back-office tools, that ratio sits somewhere between 8% and 20%. Build that. Ship it. Let the team use it exclusively for 30 days before adding anything.

The instinct to expand scope beyond the required ratio at the build stage is almost universal and almost always wrong. It takes two forms.

The first is "while we're building this anyway." A feature gets added because it seems cheap to include now and expensive to retrofit later. This reasoning is sometimes correct for infrastructure decisions. It is almost never correct for features. Features that aren't required have no validated demand. Building them before you know whether they'll be used means building them twice — once to ship, and once to fix after you see how people actually behave.

The second is "we'll need this eventually." Maybe. Build it when eventually arrives. The version of the feature you'd build today, before you've seen how the team uses v1, will be the wrong version. Wait.

The budget ceiling from Section 4 — two years of subscription cost plus 20% — applies to the full required scope only. If building the required scope already pushes against that ceiling, there is no budget for additions. If there's room within the ceiling, resist the temptation to fill it.

Maintenance cost

Build cost is a one-time number. Maintenance cost is forever.

Budget for ongoing maintenance at a minimum of 0.25% of your total build cost per month. On an 86,000 PLN build, that's 215 PLN per month — a number that sounds trivial until you've carried it for three years and added it back into the total cost of ownership calculation you built in Section 4.

More importantly, maintenance cost is not primarily a server and infrastructure cost. It's an engineering time cost: the hours spent fixing bugs, updating dependencies, handling edge cases that emerge in production, and making the small modifications that any living system requires. That time needs to come from somewhere. If it's unplanned, it comes from whatever else the team was supposed to be doing.

Define before you start: who owns maintenance, at what capacity, on what terms. If you're using an external partner for the build, establish explicitly what support looks like after handoff — what's covered, what triggers a new engagement, and what the response time expectations are. Leaving this undefined means discovering the answer at the worst possible moment.

The bus factor test

The bus factor is a blunt instrument: if the person who built this gets hit by a bus tomorrow, how long before someone else can ship a fix?

The answer this test is looking for is one working day for a competent engineer who didn't write the original code.

If the honest answer is longer than that, the architecture is wrong. Code that only one person can modify safely has structural problems: insufficient test coverage, undocumented business logic embedded in implementation details, dependencies that aren't managed explicitly.

This matters differently now than it did five years ago. AI-assisted engineering has genuinely changed the equation — a well-structured codebase with good documentation can be understood and modified by an engineer using modern tooling far faster than the same codebase before these tools existed. That's part of what makes custom software viable for internal tools at a scale it wasn't before. But "AI can help someone navigate this" is not a substitute for "this is well-built." The bus factor test is still a valid proxy for architecture quality. Run it honestly — ideally by having someone who wasn't on the build attempt a small modification before you declare the project done.

Timeline calibration

Before the build starts, map out what a minimal version, an intermediate version, and a full version of the tool look like — and agree on which one constitutes v1.

The underlying principle is straightforward: get the tool into real use as fast as the scope allows, because real use surfaces problems that no amount of planning anticipates. The most common source of scope creep isn't ambition — it's delayed feedback. The longer the team waits to use the tool, the more requirements drift based on what people imagine they'll need rather than what they discover they actually need.

What "fast" means in practice depends on the complexity of the scope and the team doing the build. A rough internal tool with no integrations might be usable in four weeks. A replacement for a system with multiple integrations, data migration requirements, and several user roles might reasonably take twelve. The specific timeline matters less than the principle: agree on the smallest version that covers the required ratio from Section 2, ship that, and let real use inform what comes next.

A v1 used exclusively by the team for 30 days will tell you more about what to build next than three months of planning. It also gives you a natural checkpoint: if v1 holds up under real use, the case for continuing is confirmed in practice. If it doesn't, you've found out before spending the remaining budget.

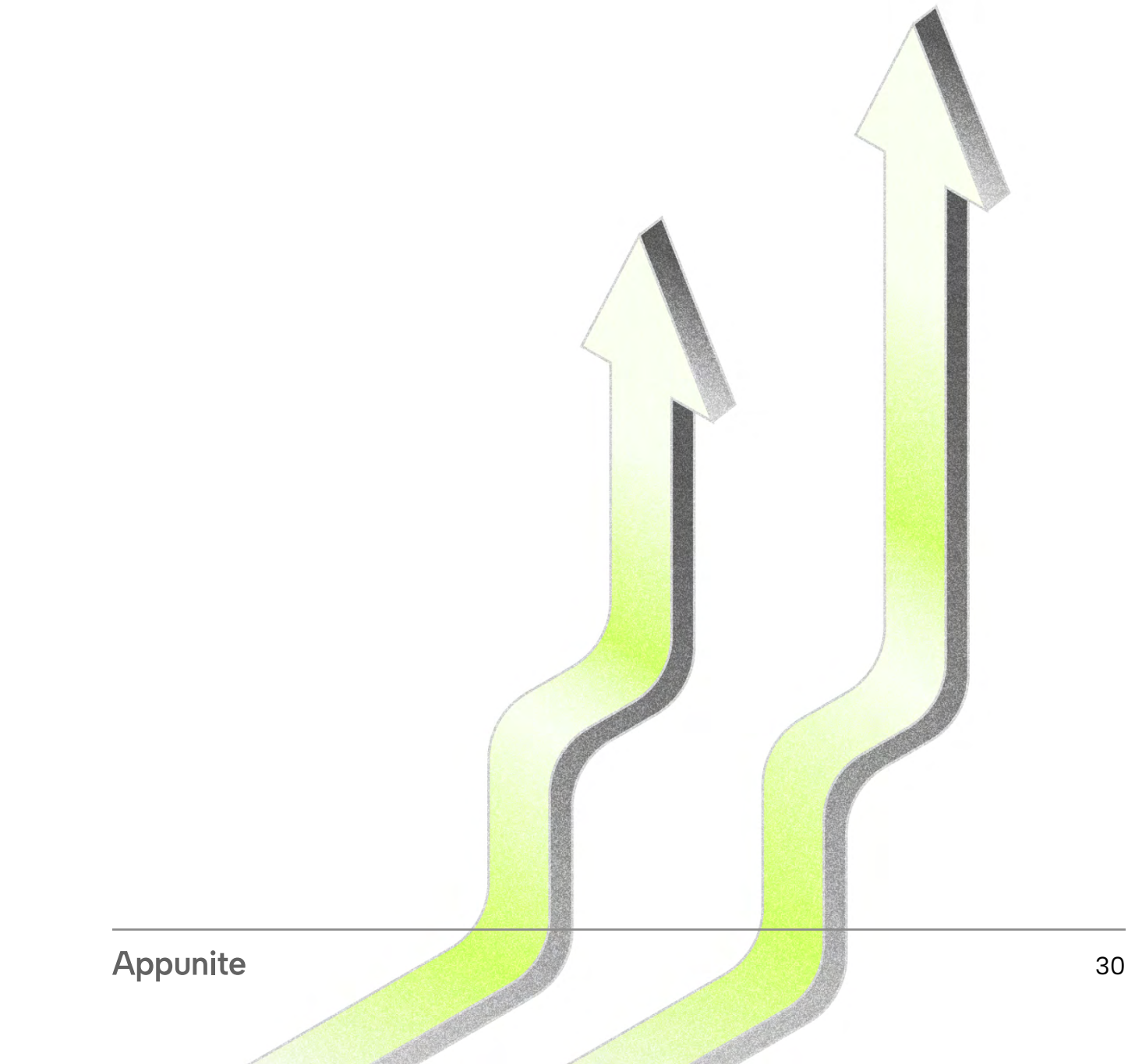
One practical question to answer before the build starts: what is the migration plan? Specifically, what data needs to move from the current tool to the new one, in what form, and who validates that it arrived correctly? Data migration is the most underestimated cost in tool replacement projects. It delays launches, produces subtle errors that surface weeks later, and often requires more time than building the features it's meant to support. Plan for it as a discrete piece of work, not as the final hour of the project.

Section 7

Our Numbers

The previous sections are a framework. This one is what it looks like when you actually run it.

We're partway through this process ourselves. At the time of writing, we've completed the pain point discovery and cost estimation phases. The feature usage audit and development scoping are scheduled — the workshop is in progress. We're publishing the numbers we have rather than waiting until the picture is complete, because the incomplete picture is part of the point: this is what honest assessment looks like in practice, including the parts that aren't resolved yet.



The tool and the cost

The tool we're evaluating: Recruitee, our current ATS. We've used it for several years. It works for basic recruitment workflows. It has accumulated a list of specific, recurring limitations that the team works around.

Annual subscription cost: approximately 43,000 PLN.

Build budget ceiling: 86,000 PLN (2 × 43,000 PLN). We're targeting to come in below this, not at it.

What the pain point discovery found

We documented 24 individual pain points across the team, then clustered them into 7 problem groups. Each cluster is named by its business consequence, not by the feature limitation that causes it.

#	Cluster	**Direct cost** PLN/mo	**Opportunity cost** PLN/mo	**Total** PLN/mo	**Annual** PLN
1	Reports calculate with errors	144	8,316	8,460	101,520
2	Metrics not customisable	192	-	192	2,304
3	Funnels not elastic	65	360	425	5,100
4	No competency matrices + sourcing inefficiency	503	338	841	10,092
5	No competency matrices + sourcing inefficiency	390	1,663	2,053	24,636
6	Calendar/scheduling gaps + workflow coordination	324	-	324	3,888
7	GDPR compliance gaps + manual processing	259	-	259	3,108
Σ		**1,877**	**10,677**	**12,554**	**150,648**

What the numbers actually mean

Direct costs: 22,524 PLN/year. This is what we can defend in a room with our CFO. It's the time the recruitment team provably spends working around Recruitee's limitations — real minutes, real workarounds, real cost.

It's also less than the annual subscription. On direct costs alone, building a replacement makes no financial sense. The ROI is -74%.

Direct + opportunity costs: 150,648 PLN/year. This is what we believe but cannot prove. The gap between the two numbers — 128,124 PLN — is almost entirely explained by one assumption in one cluster.

Where the debate lives: Cluster 1

Cluster 1 is reports that calculate with errors. Five separate reporting problems, all pointing to the same consequence: we can't see accurate hiring metrics, which means we can't identify which pipeline stages are bottlenecks, which sources produce candidates who stay, or which evaluation approaches predict success.

The direct cost of this cluster is 144 PLN/month — 120 minutes of manual reconciliation work. Trivial.

The opportunity cost assumption is that 50% of failed hires could be prevented if we had accurate visibility into hiring performance. Our failed hire rate is approximately 0.33 per month. At 50,400 PLN per failed hire (three months of fully-loaded average salary), 50% attribution produces 8,316 PLN/month — 99,792 PLN/year from a single cluster.

That one assumption accounts for 67% of our total estimated annual cost. Changing it from 50% to 0% turns the +75% ROI into -74%. The entire financial case rests on whether bad reporting data is meaningfully causing bad hiring decisions.

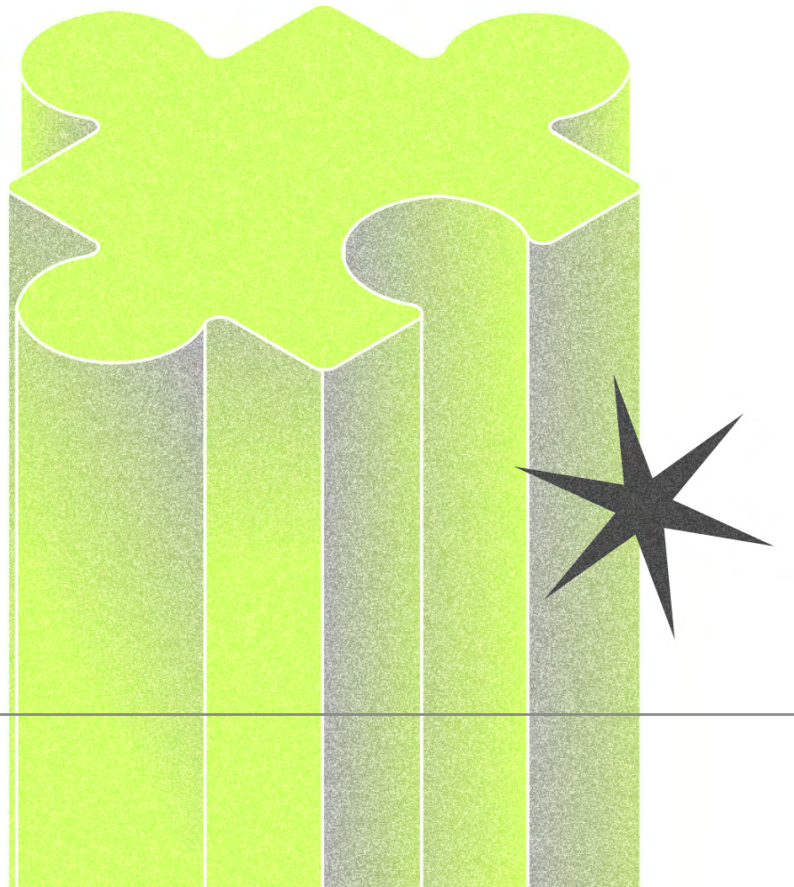
We think it is. We can't prove it. We documented the assumption explicitly and moved forward with it on the record.

The other clusters, briefly

- **Cluster 2 (metrics not customisable):** 192 PLN/month, direct costs only. The team can't build reports around the metrics that actually matter to their process. The impact is real — it prevents process optimisation — but we couldn't isolate a defensible financial consequence, so no opportunity cost was included.
- **Cluster 3 (funnels not elastic):** 425 PLN/month. The inability to edit pipeline stages after the fact causes data corruption. One problem in this cluster was rated 5/5 se-

verity — genuinely blocking. The opportunity cost (360 PLN/month) is conservative: estimated rework time per process, not downstream hiring impact.

- **Cluster 4 (no competency matrices + sourcing inefficiency):** 841 PLN/month. This is the cluster with the clearest strategic implication beyond cost. Without competency matrices and automated relationship reminders, every new role requires fresh sourcing rather than drawing on past candidates. The direct costs are the highest of any cluster (503 PLN/month) — manual LinkedIn entry, CV scanning, and candidate management that should be systematic. The opportunity cost (338 PLN/month) assumes a 20% candidate reuse rate if the tooling existed to support it.
- **Cluster 5 (no interview transcription + manual feedback):** 2,053 PLN/month. A 10% attribution on failed hires — more conservative than Cluster 1, because documentation quality is one of several factors in hiring outcomes. Still produces a meaningful number because of the base cost of a failed hire.
- **Clusters 6 and 7 (scheduling gaps, GDPR):** Direct costs only, 324 and 259 PLN/month respectively. The most defensible numbers in the analysis — pure time savings with no attribution assumptions. Cluster 6 includes three problems rated 5/5 severity, which matters for daily frustration but doesn't move the financial picture significantly.



The honest position

The direct cost case is negative. Building a replacement to save 22,524 PLN per year, against an 86,000 PLN build cost, takes nearly four years to pay back. That's a weak financial case by any standard.

The opportunity cost case is positive — but only if at least 50% of the opportunity cost estimates are accurate, which we cannot currently demonstrate. We've run the pain point discovery. We're in the process of running the validation checks described in Section 4. We know what assumptions we're relying on.

We're proceeding anyway, for three reasons we've been explicit about internally: the project has learning value (testing what AI-assisted engineering can actually deliver at this scope), it has content and awareness value (the "building in public" campaign around this assessment), and it's a proof of concept for one of our strategic theses. None of those are in the ROI calculation. They're real, and we'd rather say so than pretend the numbers justify themselves.

If we were advising a client in this position, we'd say: run the validation first. Pull the last 12 months of hiring data and look for a correlation between reporting gaps and failed hire outcomes. If the signal is there, the case is strong. If it isn't, decide whether the strategic value is sufficient to proceed without a financial case — and make that a conscious decision, not something that happens because the spreadsheet got optimistic.

What's still open

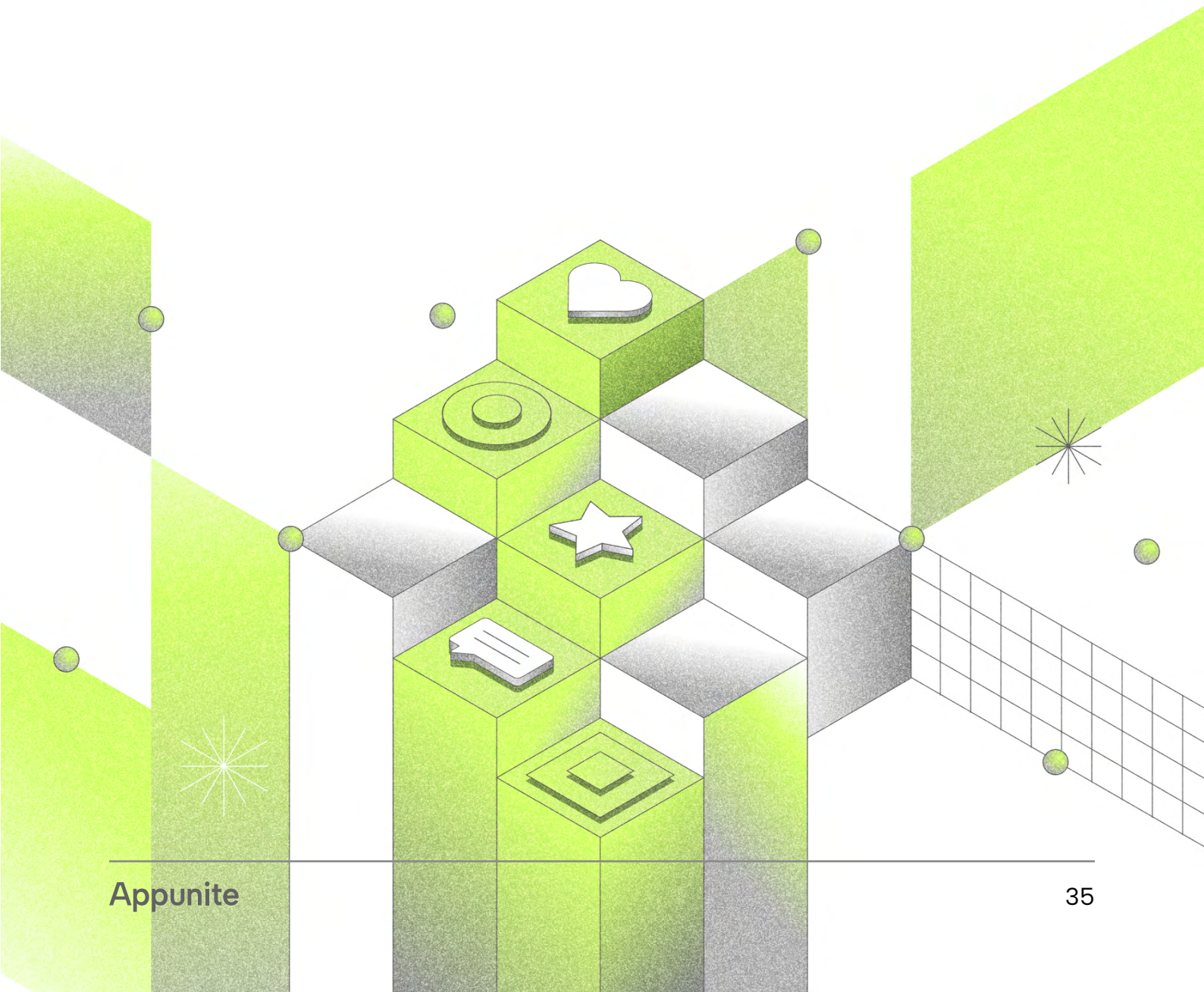
- **Feature usage audit:** Not yet completed. The development scoping workshop is in progress. Until it's done, we don't have the required ratio that would define v1 scope or confirm the "10% of features" benchmark.
- **Solvability filter:** Partially complete. We've assessed which clusters require custom logic or data ownership (primarily Clusters 1, 4, and 5). We haven't formally evaluated alternative ATS tools against the specific pain points — a gap in the methodology as described in Section 3.

We'll be posting updates in the newsletter mainly, and some of the findings will find their way into our blog and social media.

Appendices

The sections above describe how to run the assessment. The appendices are the actual working tools — duplicable Notion pages you can copy into your own workspace and fill in directly.

Each appendix maps to a specific section of the assessment. They're designed to be used in sequence, with the output of each feeding into the next. Run them with your team, the value is in the discussion they force, not in the templates themselves.



Appendix A **Feature Audit Worksheet**

What it is

A structured table for cataloguing every feature in your current SaaS tool, classifying each by usage frequency and requirement level, and calculating your usage and required ratios.

When to use it

Before the pain point session (Section 2). Takes roughly an hour with 2–3 daily users. Share in advance and fill in together.

What it produces

Two numbers — your usage ratio (how much of the platform you actually engage with) and your required ratio (the minimum footprint a replacement needs to cover). The required ratio becomes your v1 build scope in Section 6.

Sample:

#	Feature	Usage Frequency	Requirement Level	Notes
1	Candidate pipeline view	Daily	Required	Core workflow
2	Email sequence automation	Weekly	Nice to have	Workaround exists
3	Video interview module	Never	Not needed	Never activated
--	--	--	--	--
Metric	**Result**			
--	--	--	--	--
Total features listed	**87**			
Features used at least monthly	**11**			
Usage ratio	**12.6%**			
Required ratio	**8.0%**			

Open full worksheet (duplicable)

Appendix B **Pain Point Discovery Pack**

What it is

Two templates for the 90-minute pain point discovery session — a pre-work collection form and a solvability filter worksheet.

When to use it

The pre-work form goes out to participants at least 24 hours before the session (we sent ours a week ahead). The solvability filter runs in the final 20 minutes, once pain points have been discussed, scored, and grouped into clusters.

What it produces

A list of pain point clusters named by their business consequence, each classified as a build candidate, a switch candidate, or a process problem. Only build candidates proceed to the cost estimation.

Sample:

Pre-work form (excerpt):

#	Description	Frequency	Severity	Workaround	Time min
1	Hire count in dashboard always shows 0%	Daily	4	Manual export to spreadsheet	30
2	Can't edit pipeline stage after candidate advances	Weekly	5	None — data stays corrupted	--

Solvability filter (excerpt):

Cluster	Software or process?	Different SaaS solve it?	Requires data ownership?	Verdict
Reports calculate with errors	Software	No	Yes	Build candidate
Onboarding checklist gaps	Process	--	--	Fix the process

Open full pack (duplicable)

Appendix C **Cost Estimation Worksheets**

What it is

Two templates for the 90-minute pain point discovery session — a pre-work collection form and a solvability filter worksheet.

When to use it

After the pain point session, using only the clusters marked as build candidates. Run them in order. The honest split is the table you bring to the decision-maker.

What it produces

Two ROI figures (direct costs only; direct plus attributed opportunity costs), the payback period under each scenario, and the break-even attribution threshold — the minimum percentage of opportunity costs that need to be real for the project to break even in year one.

Sample:

Honest split (excerpt):

	Column A **Direct costs only**	Column B **Direct + attributed opportunity costs**
Annual savings	22,524 PLN	150,648 PLN
Build budget	86,000 PLN	86,000 PLN
ROI	-74%	+75%
Break-even attribution	N/A	50% of opportunity costs must be real

Open full worksheet (duplicable)

Appendix D **Decision Matrix**

What it is	When to use it	What it produces
A one-page worksheet that consolidates the four decision tree steps from Section 5 into a structured, facilitated output — including the final build / switch / stay decision with the key assumption and a named condition for revisiting it.	In a final 60-minute session with the decision-maker present. Not a document to fill in alone and present — the decision-maker needs to be in the room while the steps are worked through.	A documented decision, the assumption it depends on, and an explicit trigger for revisiting the decision if that assumption turns out to be wrong.

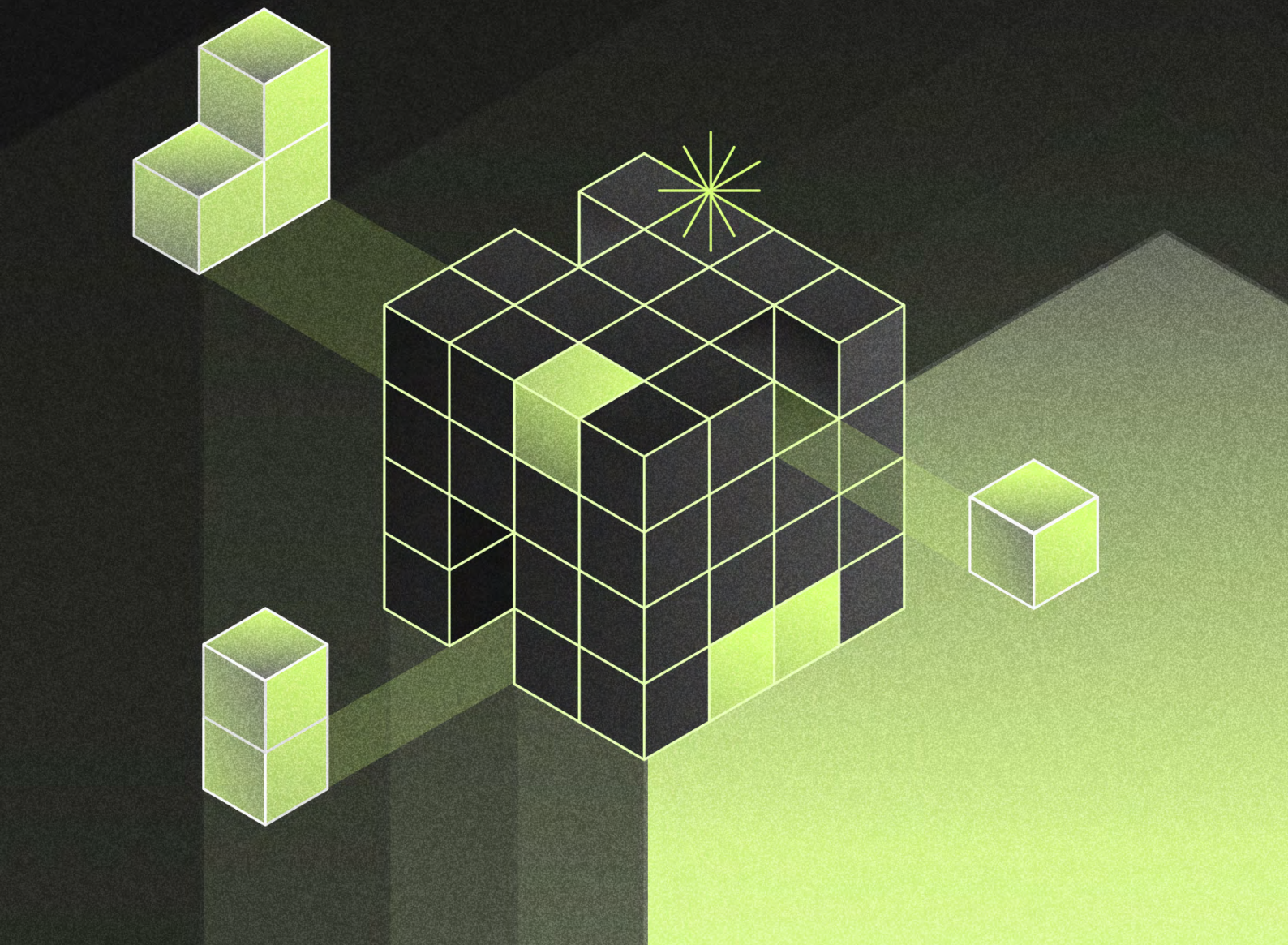
Sample:

Step	Question	Answer
1	Majority software or process problems?	Software (5 of 7 clusters)
2	Could a different SaaS solve the core pain?	No — evaluated Lever and Greenhouse; neither solves Clusters 1, 4, 5
3	Requires something structurally unavailable in SaaS?	Yes — data ownership for candidate relationship tracking
4	Financial case holds under scrutiny?	Column B only; break-even at 50% attribution

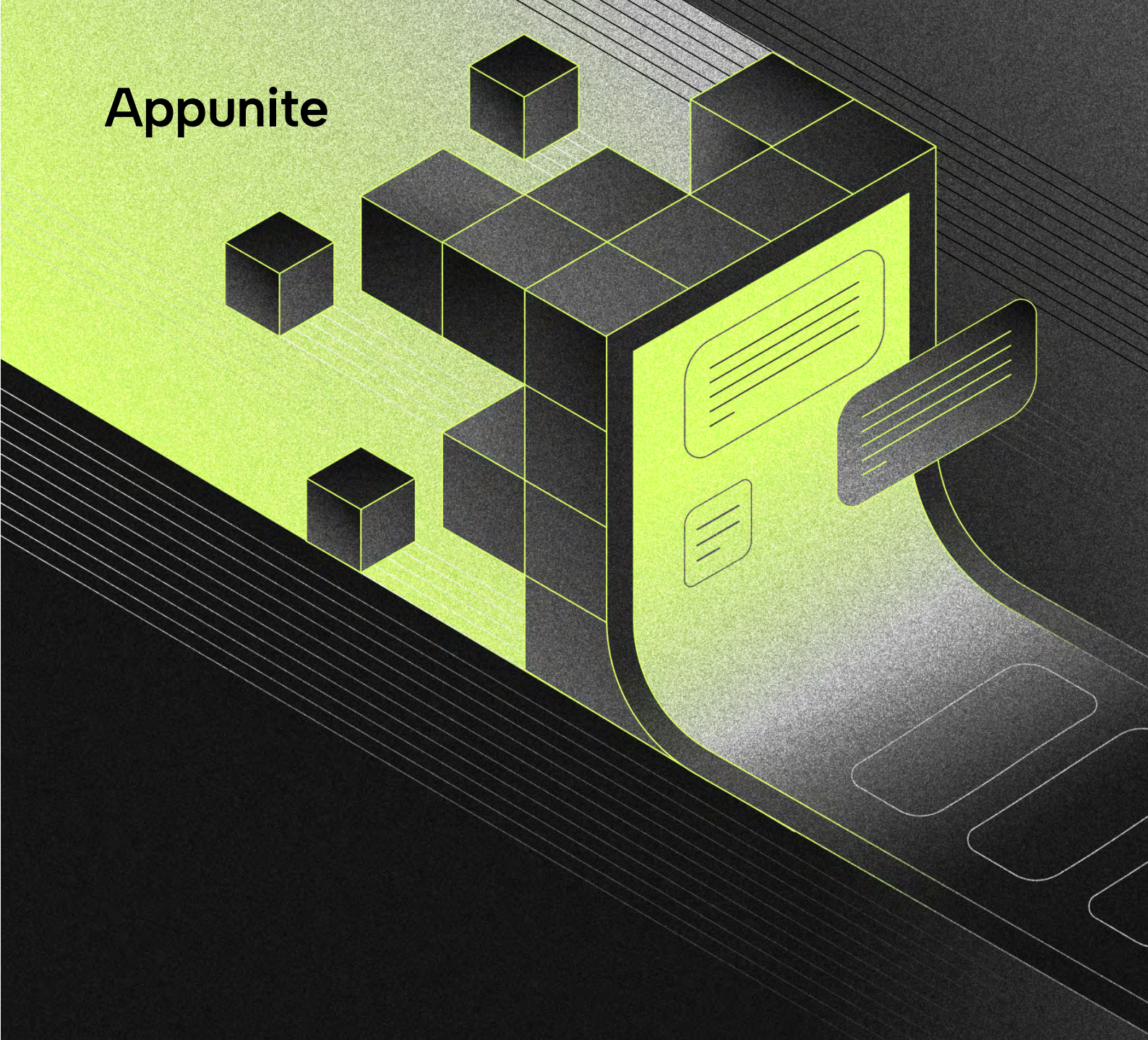
- **Final decision:** Build
- **Key assumption:** At least 50% of estimated opportunity costs (primarily Cluster 1 failed hire attribution) are real.
- **Revisit if:** Failed hire rate does not decrease after 6 months of accurate reporting data in the new system.

Open full matrix (duplicable)

Full List



Appunite



And if you want to discuss any of the above just reach out!

Talk with us at Roam →

Follow along via newsletter →