The Technical Paradox

Table of Contents

Foreword

This began as an exercise to consolidate my own understanding of a complex side effect of successful growth. Organizations are strongly incentivized to eliminate friction over the long term, and yet so much pain feels avoidable or self-imposed. Every organization that survives expansion over time will hit a point where they must choose to unwind some deeply-integrated structural, strategic, or technical decision in favor of a new path. Differing opinions on which key choices aren't working anymore and when to tackle difficult problems will rarely come into conflict at a convenient moment. Effort spent mitigating self-inflicted damage is doubly wasteful, triply if you include damage to morale, and I believe it can be minimized by sharing responsibility for the practical and realistic translation from ambition to execution across an entire organization.

My observations are from a decade as a software engineer rising to manager (and director in a pinch) with primarily two venture-backed companies, joining as early as seed stage and remaining through post-acquisition integration. I've been in the engine room running too hot and too cool for a wide variety of reasons, and enjoyed objective success with a hint of regret for unfulfilled potential.

I will frequently oversimplify functional roles to a technical/non-technical binary. This abstraction is not meant to diminish the complexity of nontechnical roles, which all have their own unique challenges as well, and is only used to paint a sharp contrast.

While I believe my abstract argument can be generalized to any technical product, I won't pretend to have experience with large-scale implementations outside of fast-paced high-growth software engineering. I've tried to remain as tech-agnostic as possible for the philosophical bits, but illustrative examples will have to focus on software-specific topics using BuzzyBee, an imaginary organization selling AI analytics for beehive owners. I have drawn on my personal observations, but they have been adjusted and exaggerated to highlight specific points; any similarities to people or software, living or dead, is purely coincidental.

The question everyone should ask themselves of the following philosophy, and all others, is: what are the specific constraints I face and how might they conflict with best practices or simple assumptions? The (tragically) correct answer to almost any question posed to an engineer is: it depends! I hope that keeping this in mind as you read on will help provoke questions rather than prescribe answers.

The Paradox

An organization that derives value from a product with any kind of technical component must confront a paradox. As demonstrated in software development, technical effort is most efficiently applied furthest from the point where the benefit is realized; it's easier to fix an error with a pencil than a hammer. However, a product must leave the whiteboard before it can complete the circle of innovation by funding further development. At the seedling stage, the benefit of technical effort is simply self-preservation by proof of an idea as soon as possible. Otherwise, a perfectly good rocket ship might never make it off the ground. It's easy to feel both efficient and agile in this environment because success is measured in survival, future be damned! A small team can sketch out a large idea on a blank canvas, and survival instinct multiplied by the euphoria of infinite possibility becomes a runaway train of technical decisions. When success arrives, typically as injected capital, it comes with new goals and constraints and technical organizations should quickly recalibrate their strategy and tactics to match the new challenges. By surviving over time, the path to success becomes more clear and they must consider which choices they made to survive until now will become obstacles. This pattern of developing technology or products, securing capital, and refining strategy repeats many times over an organization's life; each cycle is an opportunity to choose a new balance point between short-term speed and long-term efficiency.

Technical products are additive and intertwined. Each component has strengths, limitations, dependencies, and eccentricities which quickly render an aggregated technical product unique. Every choice will affect future choices, to varying degrees, and foundational choices are likely to be the most painful to unwind. Compounding this, as a product scales up and time passes, the oldest technical choices are more likely to be found inefficient because they were made with the least clarity of how they would be used in the future. Few components of a complex system can handle all future cases, scale to infinity, perform self-maintenance, and upgrade themselves. Each addition to a product introduces new constraints and complexity, often called technical debt, and can be quantified as a future operating cost plus risk. Despite the baggage the term carries in the context of software engineering… "technical debt" remains an accurate description of the accumulation of the unintended effects of technical effort. The side effects of technical debt aren't exclusively technical; an organization can become distorted and inefficient if forced to mitigate weaknesses of a product.

Some debt is unavoidable, as in cases of survival, but more often it is chosen by an organization for valid reasons. Just like financial debt, technical debt represents a gamble that an immediate outcome is worth more than a deferred but typically increased cost. This can be a good strategic choice and all healthy organizations must learn how to manage their debts. <u>The critical difference is that where financial debt is borrowed money, technical debt is borrowed time.</u> Any choice that saves time by conceding quality will confer additional risk and/or difficulty to future work in the same area of a product. Make no mistake, it is normal and healthy to carry some debt as a result of failed experiments or to accelerate growth! Strategic financial and technical debt can both amplify growth more than the cost of the debt, and both become equally unhealthy when ignored. In the case of technical debt, it can be tempting to skip paying it down because the carrying costs aren't easy to compute and root causes are opaque. Furthermore, technical teams don't always formulate strong non-technical arguments for future effects and blast radius of debt or worse, muddy the waters by labeling their own failures as debt. Over time, debt can distort an organization by gradually raising the criticality of low-value outcomes and forcing decisions which over-invest in solving the wrong problems. This is a root cause of a culture that distrusts management. Once an organization falls behind in this way they must work twice as hard to get back on track, inevitably slowing growth.

Just as carrying financial debt would be punished by compounding interest, <u>technical debt compounds by reducing efficiency of effort.</u> Coupled with pressure to meet a short-term goal, technical teams are forced to work harder to deliver less long-term value. This pressure can be induced by increasing technical capacity too slowly to meet demand as well as by becoming too complicated too quickly without reinforcement. Eventually, deferred annoyances will suddenly become blockers and unexpected work will creep into every project, delaying key features or shifting even more debt around. Technical teams will spend a larger proportion of their time doing maintenance, or other required work that keeps the lights on without advancing any particular goal. Similarly, problematic areas of a product will throw off a steady stream of issues that drag technical staff away from immediate objectives. More financial debt than ability to pay it consumes all capital and triggers bankruptcy. <u>More technical debt than ability to correct it consumes all of an organization's time and halts progress.</u> If the misery of drowning in debt while being co-signed on for even more leads to the exit of staff most critical to addressing the problems, the organization is doomed.

The currency of technical debt is time because <u>deep technical expertise, especially domain-specific knowledge, takes time to acquire</u>. Large problems require multiple people, and technical teams take time to develop a shared understanding of what they're solving for. Complex products are all unique, so even the best technical staff will require inestimable time to perform major changes to an operational product they're not familiar with; never undervalue practical knowledge gained by evolving with a product and maintaining it. <u>Continuity of shared expertise in a specific product area significantly increases success and speed by better translating organizational goals to viable</u>

technical outcomes as well as avoiding traps. This also helps an organization repay technical debt more strategically by, for example, working on faulty components adjacent to current objectives. One of the best ways to build expertise is by having a team work off some technical debt!

Veterans of scrappy startups on tight burn rates will recognize the pressure to deliver minimum functionality yesterday, with symbolic-at-best concern for long-term viability and dismissive acknowledgment of the future effort required to maintain, monitor, and scale the system. Technical staff forced to rebuild a component before deriving enough value from it will gripe: "there's never enough time to do it right, but there's always enough time to do it over!" This is fertile ground for a culture of furiously bolting components together to reach as high as possible before the product tips over, and isn't necessarily recoverable. Failing to adequately understand technical debt before acquiring it is just as unwise as agreeing to financial debt with unknown terms. Digging in their heels to resist the impact of bad technical debt, technical staff might demand perfect requirements, de-risk projects by over-prototyping, and disproportionately escalate annoyances. While these are all shadows of the solution, they create frustration for everyone and are ultimately counterproductive because they lose the nuance that some technical debt is good, and that some features can be suboptimal because they're not strategic despite their criticality. Deliberately choosing how much debt and where it lives should be an obvious goal for all organizations.

Do not despair! Balancing the terms of debt against importance to future organizational goals is a central conflict in technical organizations, but if resolved… agility and efficiency of a small organization can be preserved at later stages of growth, with the accompanying effect on profit margin and valuation. Instagram was worth $1billion with 13 employees operating under viral load at global scale. Minimizing friction will allow an organization to approach maximum potential. An organization must evolve plans that anchor realistic short-term outcomes to clear and shared strategic objectives. When strategic objectives change, so must any current plans in lockstep with internal goals and external expectations.

Every organization will be on its own unique journey, but there are at least 4 key questions that must be answered along the way. In a startup backed by venture capital, each question's answer is typically required to unlock the next round of investment and phase of growth and represents an inflection point in the organization's life:

- Is it possible?
- Who will pay for it?
- How many will pay?
- Will it generate value?

Technical debt is a recurring theme throughout all phases, but not the only type of decision that reverberates in an organization or product. How tradeoffs evolve alongside a growing organization cannot be addressed further without putting them in context of a more specific moment. Subsequent chapters will address the varying balance of short-term pressure against long-term success of a technical organization, as well as explore how technical organizations must adapt in the face of growth and success.

Example: Hooked on a Framework

A hot new startup, BuzzyBee, has developed AI that ingests sensor data and produces reports with suggestions regarding beehive health. The idea for the company spun out of some interesting findings during post-doctorate lab work, so the founders are just starting to figure out how users will want to consume and operationalize this data. BuzzyBee knows they will be doing a ton of experimentation, and must be correspondingly agile and flexible while making data available in a backend API for a mobile application.

They found a great open source library, InstantAPI, that can auto-generate the backend APIs and documentation! The CTO has played with it a bit enough to be satisfied that all you have to do is point it at a table in a database, and InstantAPI generates endpoints to create, read, update, delete, and filter the data. It's obvious this will save BuzzyBee a ton of time by letting them focus on their users and not waste time plumbing data from backend to frontend.

This is a rather reasonable technical choice for a prototyping startup, and takes advantage of a neat trick: auto-magical translation of a database table's schema to a functional set of backend API endpoints. Frameworks or tools that abstract details and leave the developer to fill out boilerplate, like InstantAPI, are incredibly powerful and can help a small team produce a ton of functional output by leveraging other developers' clean solution to a generalized problem. However, technical teams should always be wary of reliance on a framework's happy-path solution; by the time the product reaches a level of complexity which requires a solution the framework doesn't support, how deeply integrated will the framework be in the design and architecture of the product? Which other components and frameworks have been designed around the assumption that all API endpoints will work like InstantAPI does? Have developers figured out how the framework works under the hood and what might break it, or do they use the features that work well and ignore the rest because everyone was excited about a demonstration? How many deadlines on the roadmap rely on copy-paste-find-replace of similar InstantAPI endpoints and can't afford either a deviation from the pattern due to new complexity of a feature, never mind a delay to build a better pattern?

Reliance on InstantAPI is hard to escape if the cost of researching and establishing an alternative remains higher than the cost of losing time on the next new feature. As with all forms of technical debt, it will cost increasing amounts of time to pay it later; when does the benefit of taking the debt outweigh any individual feature? If development of many new API features are hindered in some small way by a framework, then for a large enough organization that will be the same amount of effort to fix the problem. This may be the tipping point in cost-benefit, but for a framework embedded deeply enough, the loss of efficiency during heavy growth can outstrip the early benefit.

The solutions to BuzzyBee's InstantAPI problem are numerous, but they all take time; what if the damage is already done? With a data model and query pattern locked to a database schema, complex queries against millions of beehives' data will slow down; first at the extremes with the heaviest users, then most users for short periods, then all users at once. Developers that have only used InstantAPI might not have a replacement ready to go, and will require time to research a new approach to solve the specific problems BuzzyBee now faces; how will this change to a fundamental component cascade into other components? Quick fixes may stem the bleeding, but how many poorly tested band-aids will be necessary under greater and greater scale? The more critical a problematic framework or tool is, the more likely a deficiency will torpedo forward progress of an organization for a few quarters. Bees deserve better.

To estimate the problems of a framework, look first to the details that are abstracted away. InstantAPI detects tables of data, the datatypes of tables' columns, and links between tables from a well-defined schema, then exposes all of it in largely the same way. Take an adversarial mindset and ask, what would be difficult for this framework to handle? Imagining some of the paths that the product could take, do any features rely on that operation or datatype? BuzzyBee doesn't know which user events will need tracking so they might decide not to define a schema for event tracking, and instead let each event type use JSON column types as it sees fit. InstantAPI, designed for prototyping and expecting mostly integers and text, might not provide the same JSON filtering operators offered by the database without extra complexity. At first, this isn't so bad because the power of the database can brute-force through hundreds of users and thousands of events. After millions of users and billions of events over years, will the same queries still be performant?

If scaling up is the trigger for a problem like this, the pain will be significant. While the balance of when to sacrifice convenience for efficiency is always shifting, <u>the best early warning signs will come from the technical team using the tool or framework every day.</u> Look at the common causes of delays in active development and listen to where they overlap with the grievances of technical teams. Monitor and parameterize what light and heavy usage looks like, and observe how failures cascade through components. To solve a problem, the organization must first understand and describe it.

Is it Possible?

The smallest seed is at the greatest risk. Few outside the organization understand the vision, and fewer believe in it. The runway is measured in months or weeks and the team can still split two pizzas. It would be foolish to pretend that anything other than survival should drive an organization's strategy at this moment. With this simplicity comes a great deal of clarity, because there's no product for a growth engine to squeeze for revenue, external investment is only looking for proof of life, and a nascent vision is not yet anchored to reality. Technical staff must seemingly answer only one question: is it possible?

Tunnel vision might help clear this gate, but why bother if it's at the expense of clearing the next? If the goal is to eventually extract value from ownership or membership, then surely it makes sense to consider how to go the distance, even at this fragile stage! Frankly, a polished demo can make anything look like a good idea, but it's worthless until sustained execution produces something tangible. Clarifying a strategy and plan to execute it is just as important as proving it's possible by developing a messy prototype. These efforts should even help refine each other, because the capabilities demonstrated by the prototype should inform the vision as much as the vision drives the prototype.

The Technical Paradox tells us the pressure of survival is hostile to robust products, and without any clear sense of the long-term plan there is nothing to guide technical tradeoffs. If decisions cannot be weighed against a specific desired future, immense amounts of technical debt can collect in plain view. Under these conditions, very little output is likely to be useful in the long-term and this fact should be embraced across the entire organization. Technical staff should be free to cover a lot of ground and bump into many of the problems they'll need to solve once survival is achieved, especially scaling concerns, and be unbothered by design errors made by solving problems serially in the order they present themselves. They must also resist the temptation to optimize; overengineering prematurely can feel like a good investment, but decisions will lack useful information gained by observing over time. The resulting prototype is likely to be an awful foundation for a large product. However, by the time a prototype does its job as proof-of-life an organization should know what a better foundation would look like. Most importantly, everyone should also be in agreement on which bits of a prototype can be reused, if any, and which need to be rebuilt first.

If the technical product of this stage is effectively throwaway, then what should the organization be focused on beyond the minimum goal of survival? An organization will weaken over time if they do not take this opportunity to build a shared understanding of all challenges facing the idea, both technical and non-technical. Coalescing around a problem for the first time may be uncomfortable; it will require technical staff to give up beautiful designs for whatever the market demands, and non-technical staff must learn enough about the problem being solved to anticipate where difficulties will lie. A good plan won't be a straight line from prototype to fully-fledged product, and should acknowledge where the vision and plan is clear as well as where it isn't. Only then can technical teams understand which technical components will be strategic and which can be rudimentary, where estimates must be precise and where they must be accurate, where to invent and where to keep it simple. An organization will not survive this first and riskiest phase without forging a clear vision, but <u>to minimize future friction they must also produce a lucid technical plan for a product that can drive into market.</u>

The organization must also prepare for growth to support the strategy they're choosing. In a small organization, it is table stakes that the Why and the What are clear, but growth obliges the How to be decided more frequently by people with less institutional knowledge. Any time spent rediscovering information previously known to an organization is wasted, so an organization must decide how they will preserve knowledge gained by multiple people over (hopefully) several years. Time is still far too precious to spend updating documents alongside changes, but <u>organizing and preserving decisions alongside their justifications at a point in time will be critical for new team members to make wise decisions in the next stage of the organization's life.</u>

No matter how unified, each prototype, first technical plan, and growth strategy will have to adapt to results and reality over time. <u>Complexity is the enemy of flexibility.</u> Prototyped components are easiest to swap out when they do a simple thing well. Each milestone with dependencies in a technical plan is a risk of delay and failure. Allowing key people or roles to be overwhelmed by unbalanced growth will add a ton of friction to early momentum. <u>Minimal is usually simple.</u>

Example: Isolate and Document Technical Risk

Data models are a foundational design element, so it makes software engineers deeply uncomfortable to slap a bunch of disorganized data into a structured database. They might still find a way to make an intelligent schema out of it, but maybe not. This is a solved problem though; unstructured data sounds a lot like a schema-less database! This might also be a mistake because losing control over data quality from the first record will hinder the organization's ability to interpret product usage data… It doesn't feel great to engineer anything when there are multiple valid approaches and no way to know which will be right or wrong.

But the pressure is on, and the only wrong decision is indecision; engineers should simply make the most convenient and inexpensive choice for themselves. Let's say the team is most comfortable working quickly with a common relational database, and integrates it into the prototype. The trap is laid; it's easy to forget that a foundational choice was arbitrary once many other components depend on it. How much schemaless data will be buried in a structured database before the organization feels enough pain, and how much more effort will it take to unwind the mess? The risk is especially high for a component that affects all others, like a source-of-truth database.

BuzzyBee's CTO receives positive feedback from early demonstrations of the beehive AI, but users want to play with the data on their own instead of riding along on a screen share. BuzzyBee decides to put a copy of the CTO's personal beehive data in the cloud and record how users interact with it. This means they'll need at least three components: authentication/authorization, a backend API to filter data in the database, and storage for event tracking.

In the long term, these three types of data probably won't all share the same access pattern. BuzzyBee's CTO is especially concerned with handling real-time sensor data and a current imprecision around the data required in event tracking, but doesn't really care about solving these problems right now. They don't have the luxury of bandwidth to monitor, maintain, and optimize multiple databases anyway! At current scale, BuzzyBee shouldn't bump into any resource limits or read/write efficiency issues. So they decide to focus on unlocking the core business case of enabling user demonstrations.

Given the team's familiarity with VanillaSQL, simplicity of set up and management, and support in many programming languages, BuzzyBee's CTO decides it makes sense to use VanillaSQL for these early experiments. They stand up one instance of it and put the data from all three components in it. Stepping back a bit, they consider the vague, but still imaginable future. BuzzyBee knows they'll need different solutions for at least sensor data and event tracking later, so they make sure to put some structure around isolating each component's usage of this shared database.

The team agrees to the following principles (and writes them down) to maximize current velocity but also guard against painting themselves into a corner:
- Clearly label schema and tables used by each component
- Draw a clear boundary around each component's data
- The CTO requires a way to provide a dump of sensor data to seed the database, and then update it periodically to match a curated dataset.

    - Make sure that this operation is repeatable and resilient against duplicate data.
    - When this action becomes more frequent than weekly, discuss how to automate it and then eventually upgrade it to the permanent version. Not now though.
- The quality of the sensor data is fundamental to the application, so enforce constraints, foreign keys, etc to make sure the backend developed on top of it makes valid assumptions.
- Allow the event tracking to use flexible JSON data types, keep it simple and let the backend decide how to manage this data for now.

A small team of people with experience using a tool or technology can likely produce a plan acknowledging nuance like this with a few hours of meetings and a whiteboard. They could also get to a similar outcome by having an engineer just go for a simple solution, but the key difference is that discussing it as a group yields a shared understanding of the tradeoffs and, critically, a written record of why the decisions were made. In a year, a new team of engineers looking to make changes to solve new problems can operate with confidence that they're improving a system that has known weaknesses rather than stumbling into a minefield of secret requirements and hidden assumptions.

Example: Core Features

BuzzyBee will need to identify its users so, like most applications, it will have some kind of authentication feature. Username/password authentication is easy to build, especially with a database already implemented in the product. The first external user demo is on Thursday; no problem! By Wednesday the team has whipped up a simple user creation and login flow, and the demo goes well.

Three weeks later, the first request for a user to reset their password comes through. The team manually updates the password, and decides they must add a way to automate this process. Again, no problem! The product was going to need a way to send automated emails to users anyway, and the team adds a "request password reset" feature to the login flow easily. Later, when a new team member goes to add the automated reporting feature they discover that the implementation of email cuts corners; this makes sense given the simplicity of the only existing requirement. However, the developer must take extra time to decide if the new features should work with or replace the existing one and thus an afterthought in the authentication feature has caused extra time in the reporting feature.

Three months later, a forward-thinking user sees the value in the beehive AI and negotiates a reduced price in exchange for feedback as a beta tester. The first customer! As part of drawing up a contract, BuzzyBee's lawyer advises adherence to a common security certification which requires an audit. The audit takes a week or two, and finds another week or two of work with low apparent value to the product. The time invested in upgrading the authentication system has already surpassed the time it took to build it.

Three quarters later, a paying customer wants to give read-only access to an intern on their team. The authentication system has never provided granular authorization, but this seems like a feature that the highest-paying customers with the most beehives will likely also want to take advantage of. Given the adjacency to security, this is something that should be done absolutely correctly and the team wants to replace it with a commonly used third party component. BuzzyBee is already sacrificing delivery of another feature to design an authorization feature that meets customer demand, so it's not tenable to consider losing even more time to replace it at this time.

Three years later, BuzzyBee is a market leader in beehive analytics but the authentication system is struggling to keep up with the load. Optimizations are stacked on top of optimizations inside the auth service, with no clear indication of when the well of hackarounds will dry up. This poses a huge and likely intolerable risk of sudden interruption to the product. Luckily, a collapse is preempted by other factors. The organization wants to split customer data across multiple clusters to alleviate scale issues in other components, but authentication is designed as a singular source of truth and would cause chaos if copied across clusters. The data migration to a new system will be difficult and intricate, but finally BuzzyBee decides to invest in using a third-party auth service that can be shared across multiple clusters. Using a better auth technology in a different architecture ends up solving multiple performance issues across the entire application, and BuzzyBee celebrates a pyrrhic victory in a battle they could have avoided completely.

It should have been obvious from the beginning that authentication and authorization are critical features; 100% of users will use the authentication features for 100% of their product interactions. If auth wasn't such a boring, table-stakes feature, it would be obvious that it should be the first component in an application to be future-proofed. In survival mode, this seemed like such a reasonable decision to do a quick-and-dirty implementation, but was it worth it to skip consideration of long-term design goals? Could that first user demo have waited a week for the team to consider the limitations of a prototyped component and budget replacement into the plan? At the other extreme, how long would the first user have waited before they lost confidence in BuzzyBee and moved on before the bulletproof authentication solution?

Who will pay for it?

Armed with a vision, a plan, and some investment, an organization is ready to increase headcount as it seeks product-market fit. This is a key challenge for technical groups because technical expertise takes time and a large influx of new team members require support to become effective and meet a correspondingly increased demand for product features. Simultaneously, for the first time technical teams will need to stop bolting onto the prototype and consider that eventually the organization will value stability and scale more than completing current projects. On one hand, the organization will choose to build components of the product that should be permanent and thus will require precision and time. On the other hand, an organization with a plan embracing the unknown should expect chaos as the price of agile experimentation. <u>The tension between agility, stability, and self-improvement renders organizations especially vulnerable to themselves while grasping for product-market fit.</u> Any unresolved disagreement about which tradeoffs to make will be most damaging at this moment; technical teams will mismatch time invested with strategic importance and produce flimsy foundational components or over-engineer minor features. These issues will be amplified by organizational growth, as new team members inherit disagreements without understanding their source or having the power to resolve a conflict.

Externally, success in this phase is simply provable product-market fit. However, long-term success will require the ability to improve and deliver the product at scale. If an anticipated or confirmed product-market fit can't be delivered at scale to meet growth expectations, what's the point of building a product to fit that market? No doubt this is a moving target, but as an organization matures through this stage the target should begin to stabilize; <u>an organization must acknowledge that their goals are still in flux, and spend precious time efficiently by working where the goals and requirements are currently most clear</u>. Even where goals aren't clear but time must be invested anyway, peering beyond the horizon can still provide focus for immediate goals if it narrows the scope for a long-term technical solution.

Given a technical roadmap informed by the prototyping phase and some feedback from the market, an organization can begin to invest time more wisely; it is critical to spend an amount of time designing a component commensurate with the longest possible view of its lifetime. That is not to say that all aspects of the design must be built, but if a component is the axle to a product's wheel, then it should either be future-proofed for anticipated growth or designed to be easily swappable for a better version when necessary. Conversely, if a component is there just to check a box but isn't key to the product or growth, then very little time should be spent imagining the far future of it. These choices aren't always aligned with the simple or instinctive choice, so the Why is increasingly important. Unlike prototyping, <u>most of the choices made in this stage will reverberate for years so technical debt must be chosen extremely carefully at this point.</u> Often, the effects of these choices are subtle and rely on intuitive knowledge of the problems as well as how any existing components are solving them.

The technical paradox makes it clear that the early stages of product growth is where design choices are most impactful to the long-term outcome. This is one reason why it's important to shift the paradigm of decision-making immediately upon receiving investment. People with the most knowledge of key components can be tempted to work alone, citing efficiency and speed, but will likely continue to increase the opacity of the outcome. Similarly, handing off proof-of-concepts to a new team member can feel like delegation, but it doesn't help new team members absorb the problem enough to intuit solutions to unforeseen problems. Inevitably, sketches of ideas lack depth and no matter how good a team is at painting inside the lines they will miss unstated or unknown requirements. In this stage, <u>technical organizations can only grow sustainably to meet full potential if they prioritize developing the capabilities of the team over the capabilities of the product.</u>

Failure to do so ends with a fixed set of experts obligated to provide knowledge to a growing majority of low-capability teams. If experts are stretched too thin, management is forced to make an ugly choice far too early: should a unique expert be deployed on a single critical project but risk failure elsewhere due to lack of key information, or should an expert be spread across many projects but risk being ineffective for all of them? These options sound like technical debt because they sure are; resource constraint is the easiest reason to take debt too early and will flatten the exponential growth that this phase aims to unlock. Despite the external pressure to demonstrate product-market fit, <u>a small organization must lay a strong foundation for structure, process, and culture that will be capable of delivering the vision while they're still figuring out exactly what the product should be.</u>

Specifically, this means creating an environment in which new team members can quickly become effective, as well as guardrails that prevent them from violating key principles of components' design or their role in the long term plan. That foundation might be best laid by hiring only elite candidates, but this is a dangerous strategy to stick with because it's unsustainable at scale. If an average or junior technical team member can't be effective in the organization, then the organization's work throughput is capped by the number of perfect candidates they can find and afford to hire. Despite the appearance and discomfort of slowing down work on critical product features, investing in keeping everyone moving in the same direction will yield huge gains; <u>it is the responsibility of leadership to teach, not simply direct.</u> Only then can an increasingly large organization consistently produce technical outcomes that will save time during exponential growth instead of waste it, allowing the organization to reach maximum potential.

A common choice made at this time is to build or integrate various components. Building a custom component takes time and carries risk, but unlocks deeper customization and is generally where innovation can happen most efficiently. Integrating a third party's component isn't always cost-effective, but the abstraction of an entire problem or domain typically saves a lot of complexity and maintenance and thus is often good technical debt… but only if it can be isolated with an eye toward suddenly needing to be replaced for non-negotiable reasons in the future. Zoom out a bit and this isn't terribly different from how an organization should treat the components it builds; trusted design goals for components include separation of concerns, failure-tolerant interfaces between components, and simplicity wherever possible.

If all goes well, the organization will build their own repeatable process to deliver components of the product around about the time that customers also start to see value. This is the last good opportunity for a technical organization to set design patterns, institutionalize product testing and monitoring, and smooth the logistics of delivering the product to customers. The next investment triggers explosive growth, both of the technical organization as well as the marketing and sales engine that drives customers to the product. The exponential growth itself will also put a lot of pressure on the product, making it increasingly difficult to delve into the inner workings to address key bottlenecks or continue to innovate. With growth comes entropy, and so this stage is the last moment that an organization can exercise strong control over the technical foundation the product is built on. Any investment past this point will come with high expectation of return, and a need to demonstrate consistent exponential growth will weigh heavily against technical concerns.

Example: Optimize for Efficiency

BuzzyBee is expanding their beta program, and the marketing team has driven a few leads into the sales funnel. These potential customers look a bit different than the current beta customers; they have many more hives across more locations so they have a very different use case for the product. Despite concerns about the effort required to meet their demands, these large customers have a lot more money to spend on software services and BuzzyBee wisely chooses to explore the opportunity!

In the face of uncertain product-market fit, it is tempting to shift the focus of the technical team toward a new potential market. This would accomplish the immediate goal of non-technical teams choosing a path forward, but has the side effect of making a mess in the workshop where technical teams continue to deliver and maintain the product. BuzzyBee's technical leadership doesn't feel they have enough information about the future direction to make architectural decisions, so they accept the chaos and embrace 100% investment in innovation to see if they can unlock large customers. For a short period of time, deploying all technical resources to an extended hackathon isn't necessarily a bad idea, but it is still a survival-mode reaction to a nuanced opportunity.

BuzzyBee's mistake in this decision is forgetting that they're not working on the prototype anymore. There are a ton of problems a software organization needs to solve that aren't specific to the product; an organization must deliver the product to customers and address issues with functionality as they arise. How is it deployed to production? How are defects and accidents prevented? How will the organization know if the product is working correctly? What features will internal sales and support users require to administer the product? These are all opportunities for a technical team to make progress on projects where the outcomes are more clear because they're more generic. If designing the data/AI pipeline is impossible because BuzzyBee doesn't have enough information about the workloads it must support, then time would be spent far more efficiently building the CI/CD pipeline that will support all current and future projects, or building a common pattern for components to monitor and report their own health to proactively identify problems. Infrastructure, Developer Operations, and Business Operations all describe tons of work with simple base requirements that can make an immediate and lasting impact on the throughput of technical and non-technical teams.

Unfortunately, BuzzyBee gets excited about the new large customer segment and goes all in on chasing functionality to close these deals. It works, and a year later BuzzyBee has exceeded all the revenue targets despite many failed experiments. However, defects and downtime are a frequent occurrence because of the under-investment in infrastructure and stability supporting the development teams. BuzzyBee decides to invest in monitoring to prevent backlogs in data processing pipelines, but the cost to do so is far higher now; in addition to creating the solution, technical teams must also figure out how to inject the solution into components that may never have considered this requirement as part of their design. If ensuring visibility into functionality was a requirement from the beginning, components' designs might handle this challenge more elegantly and have saved technical effort. In the worst cases, components may not be able to adapt to a backward-applied pattern for monitoring application health; organizations will be forced to choose between even more expensive investments or skipping monitoring in problematic areas.

If BuzzyBee's technical leadership had instead resisted the demand to over-deploy teams on effort with unknown reward, they might find that solving the problems common to all software development projects accelerates research as well as reduces the side effects of experiments gone awry. New team members added toward the end of this phase will become more effective more quickly and, taken together, all these gains in efficiency could have primed BuzzyBee to enter explosive growth at a far higher speed at the cost of spending slightly less effort on new development.

Example: Think big, move fast!

BuzzyBee has passed fifty customers, and all of them rave about how predictive analytics is increasing both honey production and crop yields. A few smaller customers have left, having made big improvements from six months of product usage and learning how to do better by themselves. BuzzyBee is close to product-market fit with customers who manage 10-100 hives, and the organization chooses two clear objectives to improve the product while retaining more users.

Objective 1: Improve analytics even further from customer feedback. The newest engineering lead used to work for BigCloudCorp, so they're primed to help upgrade the sensor data collection and analytics engine. The backend team designs a real-time data pipeline that collects key beehive data with sub-second granularity, and with even more data the AI team tunes their algorithms for rapid response. Customers are impressed by the results, and cutting edge technology helps BuzzyBee close deals with larger and larger customers… until the data pipeline starts falling behind real-time one day. The newest customer is one of the largest honey producers in North America, and has a hundred times the number of hives as the next-biggest customer. The backend team is shocked, they never considered that this much throughput could come from a single customer and a pipeline component is failing. The easiest solution is to reduce the sample rate from ten milliseconds to ten seconds, 1000x slower; it reduces the data size back to what the data pipeline can handle and might even make smaller customers run faster as well… until the AI team starts reporting false alerts are firing to all customers; they've spent the last nine months optimizing the algorithms to pick up on tiny fluctuations in hive data and data points every ten seconds have a much larger variance.

Objective 2: Make the product stickier by adding hive management features so BuzzyBee becomes a key tool and not just a source of information for customers. The product team thinks a workflow management feature will drive engagement to the application; help customers assign daily maintenance tasks to their employees, who then record results and log events in the application. The best part is, it's simple to add since it’s very similar to existing endpoints making data accessible to other components! User engagement in medium-sized customers skyrockets immediately! However, the smallest customers are one or two people using pencil and paper. The largest customers have purchased other software for this purpose; the software is both already operationalized and way more feature-rich than BuzzyBee’s management tools. Six months later, BuzzyBee finds product-market fit as a pure analytics product for large customers and further development on the management feature that medium customers love is abandoned. However, the cohort of customers using it will leave if it’s removed so the organization makes the obvious choice of leaving it in place… until the database starts alerting; the write operations from legacy customers are blocking write operations from the AI engine from the influx of large, new customers.

Both objectives began with a completely reasonable idea, but cascaded into conflict between key technical components. Each choice was justifiable in the moment, so is there fault to be found in either case? Perhaps not; building a product is messy, and dead ends are impossible to avoid completely. An organization must learn to keep track of the side effects of experimentation and predict messy problems. If anything, BuzzyBee should have reacted more quickly to understand the unknown risk of assuming key components will scale in ways they weren’t necessarily designed for. Beware: complex problems with intermittent warning signs are easily ignored when they’re attached to general success.

How many will pay?

Everything is starting to click. The growth engine is cranking out new customers and current customers are increasing usage as they integrate the product more tightly. Financial projections are right on target for maximum valuation, and success is around the corner! All the organization needs to do is capitalize on this momentum as they make their largest relative gains in market share, before settling into stable growth typical of a large enterprise. The organization is simply not a scrappy startup anymore, so the product is expected to work reliably by now. The train is rolling in the right direction, and fear of screwing it up can begin to creep into decision-making. <u>Even though demands on the product are increasing dramatically, the organization's tolerance for technical issues that impede growth is reducing.</u>   The equilibrium has shifted across the pivot point, and stability is becoming more important than agility. While this is a side effect of success, it typically doesn't mean that technical staff can relax; the expectation is that growth will continue to accelerate! However, the product will experience an immense amount of pressure due to scaling up, and the worst technical debt is likely to be exposed now. Unknown debt is especially likely to come due in these moments, when expectations change more quickly than technical groups can react. Awareness of potential issues gained from clear decisions and tradeoffs in recent history will go a long way toward predicting issues and mitigating them before they become showstoppers.

In addition, explosive growth will also place a lot of pressure on the technical team. Demand for technical effort is increasing to match customer growth, and the technical team will add headcount to meet it. This poses a few new problems, most notably that the impact of an individual's technical contribution weakens relative to a larger and larger organization. Similarly, growth always (temporarily) dilutes institutional knowledge. Where a team could previously rely on a small group of experts to tackle the thorniest problems, the product is ever-growing and simply because of size and complexity there will be a higher quantity of critical issues. <u>The viability of the technical team under heavy growth is directly linked to its ability to make new team members effective.</u> The ability to absorb new people into a culture and technical product's ecosystem is an obvious, but frequently unstated, assumption of a plan to meet exponential growth in customers. Poor documentation of previous designs and tradeoffs, lack of guardrails, and lost knowledge will all be exposed. Goals missed for these reasons represent paying off interest, not principal, on technical debt of yesteryear; too much debt will be a drag on the whole organization, not just technical staff working on the product. Pay close attention to common themes of failed projects, and ask: how much unpredictability can the organization tolerate in any project that touches a problematic component? Continued growth will likely only make it worse.

Taking these organizational and technical factors together, this is a seismic shift from the previous phase; the balance of agility OR resilience suddenly becomes a mandate of speed AND stability. A technical organization unprepared for this moment risks drowning in issues du jour because they haven't invested enough in discovering the root causes, are too busy to help new team members to increase capacity, and too frustrated to work with nontechnical groups to negate problems with creative solutions. The biggest challenge for the organization is how to prioritize and allocate technical effort, because the balance has swung from risk-tolerant to risk-avoidant. Simple additional features are easy to justify to customers or non-technical teams, and can be very effective drivers of organizational goals while still feeling low-risk. By contrast, complex surgery in core components addressing technical debt can potentially solve several problems across several components all at once but is generally high risk. How can decision-makers choose well, especially when the organization's size means deciders are typically not informed by direct involvement in execution?

A primary source of difficulty is that technical teams are notoriously terrible at justifying effort to pay off debt. To be fair, the benefits can be both obvious and vague at the same time. Optimizing a known bottleneck will surely improve the system, but how much will it reduce the cost of customer acquisition or churn? Adding more testing to product delivery pipelines should reduce defect rate and make product development more efficient, but will it pay for itself in the long run? An ancient component works well most of the time, but is impossible to debug when something goes wrong; will it require less effort to research and rebuild it than playing whack-a-mole occasionally for the next few years? Technical teams can usually describe their problems and solutions accurately, but find it difficult to estimate effort and benefit; this imprecision weighs heavily against deciding to pay down debt, and it's easy to default to features or ideas with clear tangible upside. Organizations must resist the temptation to choose the simple or easily justified decision, and by now they can afford to give technical teams time to explore some problems without tangible upside.

Technical leadership would be wise to apply some fixed minimum amount of effort toward research and improvement of existing components or processes; this should be table stakes for an organization of non-trivial size. As a simple or small product, it's easy to unify a simple solution for a narrow segment of customers. As the organization grows and the product grows or becomes a suite of products, so does the breadth of customer type and thus the complexity that the organization must support. Without reinforcement and investment in the logistics that connects a large organization, they will not be able to take more market share without falling apart and losing it all. A significant ongoing amount of investment in improving connections between the product, technical groups, and non-technical groups is no longer optional under exponential growth. To prioritize this effort, technical and non-technical teams must teach and learn as equal partners. The technical paradox still applies; many of these issues can still be simplified or negated entirely at a whiteboard with a simple product tweak or minor change in organizational process, but it requires all parties to understand the problem completely.

An underrated benefit of prioritizing technical debt is happiness and efficiency of technical teams. Continued development and an increasingly complex product means debt is still being created, and that's perfectly normal! More debt means technical work is harder and less fun, but as the organization matures technical teams will expect their work to be less frantic, not more, so despite continued acceleration in product development the net amount of technical debt must stabilize and then reduce from this point forward. Otherwise, an exponentially growing organization will simply collect debt exponentially alongside customers until it splats on the ceiling. Balanced well, an organization will feel that the growth in revenue, customer usage, and headcount will directly correlate with improvements in quality and efficiency of the product and teams working on it.

Example: Bug-Free Software

BuzzyBee’s bespoke AI is truly impressive, and the technology deserves its winning share of the beehive predictive analytics market. Equally impressive, but much less visible to customers, is the underlying data pipeline that receives, aggregates, and stores a staggering amount of video, audio, and environmental sensor data. Without this system, composed of many components, the AI would have nothing to analyze. BuzzyBee’s technical team is justifiably proud of the data pipeline; they have spent years solving a complex series of problems with a strict constraint of the AI’s low tolerance for bad or missing data. The overarching design goal has been to eliminate failure, and technical teams have succeeded to the point where the only failures originate from hardware, not software.

Reliability in the face of messy and uncontrolled input is quite a feat of software engineering, but at what cost? A low tolerance for error necessitates a high degree of suspicion of input data, and corresponding rigor to detect and prevent edge cases that the software isn’t designed to handle. As a result, a higher-than-average proportion of the code in this system of components will be dedicated to defensive programming relative to the business logic that drives the outcome a customer experiences. Over several years, handling for edge cases accumulates to the point where, no matter how well designed, components have a lot of complexity buried in the details. Even though the outward appearance is a data pipeline that works spectacularly, the organization has made a number of tradeoffs that might not be obvious.

The most important side effect is that complexity and subtlety, especially if accumulated by years of minor improvements and fixes, are a huge magnet for hidden institutional knowledge. If a new bug appears, it will always be fixed most efficiently by a person who already knows how the component works. This is the most common cause of knowledge silos, which become a huge risk to the organization if a critical system can only be worked on by people who invented it or have logged hundreds of hours maintaining it. If the organization assumes some fixed cost of maintenance based on certain people being available, loss of a key person can blow up a bunch of unrelated projects by forcing an organization to divert teams off-plan for far longer than expected. The organization may get trapped in this cycle, never choosing to invest in knowledge redundancy because there's too much work to do. Lack of redundancy is a particularly sneaky form of technical debt, not because the debt itself is large but because there's only one or two people who can pay it off.

Another side effect of systems that prioritize rigor and correctness is that the mechanisms employed to guarantee these goals are typically not very interesting. Clever, sure, but working on components like this isn't typically enjoyable, especially for new people trying to learn; morale is easily killed by doing something incorrectly over and over for reasons that are almost impossible to intuit until you're aware of them. This is dangerous because it can quietly become irrecoverable; beware of technical teams suggesting a full replacement because they hate or fear working on a component. The effort required for someone new to wrangle the complexity of the old component without guidance can be extremely high with almost zero improvement in customer value. Similar to the example above, technical debt strikes again; a low carrying cost on an increasingly large amount of debt isn't so bad until paying it off becomes necessary.

All that said, it is reasonable for BuzzyBee to choose a requirement of high reliability from key components. If the pain of complexity comes from lack of flexibility, then BuzzyBee should protect themselves by ensuring that a large swath of the team understands this key system by investing in sharing the load of working on it over time. Documentation is another obvious choice, but only to the extent that it notes decisions and rationale and not an exhaustive listing of specific cases. A more difficult question is: should the AI components reliant on the data be allowed to be fault-intolerant? This would negate some of the pressure placed on the design of the data pipeline, but would the reduction in effectiveness in analytics affect BuzzyBee's market share more or less than the risk of being one nasty bug away from blowing up an entire quarter's worth of features?

Example: Weight of Success

BuzzyBee just cannot stop growing! Technical teams aren't worried about being only one step ahead of the growth because they've built out rich monitoring and can scale components dynamically to meet peaks in load. The biggest challenge is actually not the technology or the product right now, it's continuing to hit exponential growth targets. BuzzyBee attacks this problem by increasing development effort to retain current customers, upsell current customers, and the unique challenge of international expansion to find more customers. These projects come with a lot of pressure from a sales organization chasing ever-increasing expectations and only so many leads. The roadmap is full, and there's not a lot of room for error for technical teams and their management.

BuzzyBee is a mature organization at this point, with all the associated support and administration requirements. Customers expect simplicity and convenience from billing and product support. Potential customers recognize the brand, and expect a polished sales process that aligns with their confidence in BuzzyBee's good reputation. Much like the authentication and authorization features, an extremely high percentage of customers will use or benefit from administrative features; the boring bits of the product are consistently the most important or heavily used! BuzzyBee has a strong culture around the quality of the AI, their key differentiator, and has under-invested in many of the tools that connect the product to the other functions of the organization that rely on information or administrative actions. This has been fine so far because internal users doing admin or operations have learned as the product has grown, and technical teams can advise on and execute rare requests to do something unique the product doesn't support.

As the international expansion rolls out, the growth engine cranks up yet another gear. BuzzyBee starts peeling people off active projects to put out small fires around scale, and every functional group starts hiring to support the growth. More users, more leads, more customers, more, More, MORE! Of everything! Continued growth of an organization will eventually expose inefficiencies; it's easier to double headcount after 2 years and 20 people than 5 years and 200 people. BuzzyBee is about to take a big gulp of champagne problems.

BuzzyBee's AI requires a certain amount of data from each specific hive before it can perform analysis, and the delay is anywhere usually hours but sometimes weeks. Once in a while, it doesn’t work at all. BuzzyBee is aware of the environmental factors that cause this and experienced engineers can produce an estimate from instinct, but the process usually works eventually. With tons of demos scheduled as leads come in hot, the sales team is less tolerant of rescheduling demos for important or large potential customers and so they lean on engineers to help them decide if the demo will be ready in time and look good. Which feature should be delayed to support reasonable questions from the sales team?

BuzzyBee’s technical teams often create and destroy testing environments where they can deploy a full copy of the product. Once in a while, a team will brick their environment but it happens infrequently enough that the infrastructure team uses a manual override. Not only do all the new hires represent twice as many testing environments, they’re even more likely to brick something because they’re less familiar with the components. What won’t the infrastructure team be working on while they spend a few hours a week mopping up the same messes?

BuzzyBee’s admin UI in the product is pretty rudimentary, but it works well enough for creating new accounts for prospects and seeding them with some data. Veteran salespeople know to keep profiles small and light to prevent wasted effort in the backend, but the demo looks better with more data. New hires, ignorant of the side effect, use the simple admin UI to overstuff demos for new leads. Will this extra data tip the system past redline and into meltdown? What if the cleanup of a minor meltdown is also a manual process?

These examples are all symptoms of the same under-investment in the glue that holds the various parts of an organization together. Without deliberate and ongoing effort to connect teams’ workflows, BuzzyBee’s functional groups will operate out of sync and lose efficiency to the side effects of each others’ actions. The work to improve these problems typically falls on technical teams, which raises a familiar question: can BuzzyBee accurately balance the effort and reward of projects that are as different as “add this feature” and “reduce errors from manual processes”?

Will it generate value?

Riding a wave of exponential growth and racking up huge revenue is certainly cause for celebration! Building an organization that can market, sell, deliver, and maintain a product means that shareholders will almost certainly receive a return on their money or time invested. As exponential growth tapers off an organization's valuation will come from more than just predicted growth; <u>financial success is inextricably tied to the health of the organization and product because this is when technical debt starts to be felt in financial terms</u>. If an organization is trying to demonstrate a path to profitability, the number of people required to supervise and keep-the-lights-on in the product is suddenly going to become as important as new features that drive new revenue. The amount of effort to support continued customer growth without interrupting delivery of the product will put a ceiling on valuations, so it's time for the organization to reap what it has sown.

Investors are beginning to care about extracting value, not simply demonstrating it, which changes many of the rules that the technical strategy thus far has been based on. Furthermore, a successful product and growth engine doesn't necessarily require as much innovation to keep chugging along and product development yields diminishing returns. <u>There's no need for agility when stability allows a mature organization to drive profit growth by reducing new development and dominating current markets</u>. However, any strategy leveraging economy-of-scale is dependent on a product that can scale and won't impede ever-increasing sales. At this point, a product or group of products have probably been under constant expansion and development for several years, so despite any ideal of simplicity or appearance to users a product that has gotten this far is inevitably complex. No one person knows how every component works, and turnover over time has led to some complete gaps in knowledge. If a product needs to service only 50% more customers to stop losing money but some critical component suddenly can't handle the growth, the halt in progress and delay in extracting value might be measured in months or years with a corresponding reduction in the amount of value extracted.

These forces add up to a set of rules entirely different from what the organization started with. <u>Technical teams in large organizations are no longer balancing agility against future resilience, so new, different, and situationally-unique factors take precedence at scale</u>. Predictability, delivery time, cost, and a web of organizational approvals each add their own axis of complexity. A decision that adds permanent maintenance suddenly becomes tolerable if it eliminates a ton of effort to figure out what the permanent fix would be and get it approved by a huge technical organization; the incremental cost to hire staff in lower cost regions that will perform permanent maintenance is a drop in the bucket of a multinational organization's costs. Conversely, large organizations seeking efficiency by cutting costs run the risk of losing the battle with technical debt over time; a temporary boost in profit margin will erode over time as debt accumulates. Organizations should be careful when constructing incentives for decision-makers; which strategic principles should counterbalance the increasing pressure of a fiduciary duty to maximize shareholder value?

Unfortunately, a large organization's completely new set of problems and available solutions will push much of the preceding concerns out of context; <u>with large-scale success comes the expiration of a philosophy balancing survival and potential</u>. Consider instead: should a successful large organization scale back investment and growth to grind out consistent profit? Can shareholders tolerate the cost and risk of continuing to innovate?