# A Macroeconomic Approach to Scaling Software Organizations

Fasih Khatib
fasihxkhatib@gmail.com
+91 83900 84972

*Measure the process, not the engineer.*

# Table of Contents

**TL;DR**

# The Executive Summary

Scaling a software organization from its early days to its growth phase introduces several critical infrastructural and organizational friction points. As an enterprise moves beyond its initial core cohort and expands its ranks, leadership demands rigorous methods to evaluate how this expansion alters systemic velocity. This operational mandate requires measuring the exact financial return of adding headcount while gaining absolute visibility into how underlying deployment processes withstand rapid growth.

While the software engineering industry is saturated with frameworks attempting to quantify these precise dynamics, existing paradigms are structurally flawed. Many of them subject developers to a toxic micro-surveillance microscope, tracking superficial variables like closed tickets, while others demand prohibitive platform engineering runway to instrument data collection. Furthermore, the metrics designed for capital-flush big tech entities are fundamentally unsuited for a lean, growth-stage startup. This paper outlines a zero-cost, non-invasive methodology to collect high-leverage metrics that drive deliberate, system-level process optimization.

Chapter I

# Introduction

All successful software startups go through a growth phase. They start as a few engineers that form closely-knit teams and eventually grow into many engineers that form loosely-knit teams. The tools and processes that worked in the early days of the startup begin to collapse under the weight of growth. The infrastructure that once reliably deployed software now creaks as many engineers try to release their code, the new teams try to adjust to the systems they have inherited, individual engineers wonder about the trajectory of their careers in a fast-paced environment, and management tries to bring order to all this chaos.

As the organization becomes large, diversifies into various specialized departments, and tries to maintain its velocity while staying nimble, the question that presents itself is how best to keep everything running efficiently at this scale: how does the management, who is now overseeing many engineers, get a sense of what is working and what is not? The question of justifying the growing headcount presents itself, too: how does the management know that hiring the next few engineers improves productivity?

The industry has tried to answer these questions by coming up with various frameworks to measure the productivity and overall health of the organisation. These frameworks range from calculating the raw output generated by a single engineer, such as the number of lines of code they've written, to gauging the health of the software development pipeline, such as Google's DORA, to more holistic frameworks that capture both the raw productivity of engineers to the state of their psychological safety, such as Microsoft's SPACE.

Google's DORA framework is based on five key metrics based on the twin pillars of throughput and stability. They measure how reliably software goes from raw, committed code to stably released in production. A body of research indicates that these metrics accurately predict how well teams can ship code and act as indicators for prioritizing improvements and validating progress. However, these are metrics that approach the health of the organization from a very delivery-oriented view point. Specifically, they measure velocity and stability of code changes.

Microsoft's SPACE is a holistic framework that looks at various aspects of a software developer's work from their raw output to their feelings of satisfaction. In contrast to calculating one metric to reflect developer productivity, it aims to provide a multi-dimensional suite of metrics that capture everything from satisfaction to clarity of communication among members of a team. However, the question of how to capture these metrics is left completely to management, with the recommendation to choose at least three metrics from the five. Additionally, some of the metrics work at the individual level which may lead to optimizing the metric itself becoming the goal.

While these frameworks are comprehensive and useful, implementing them at the growth stage of the startup may prove cumbersome due to the complexity of the process involved in collecting these metrics. What the management needs is a quick framework that they can use to gain insight into what is working and what is not. Additionally, these metrics need to be collected on a system level, instead of an individual level, so that the focus remains on improving the process instead of the person. As the title suggests, the theory of macroeconomics provides a valuable body of knowledge that can be used in this specific context. The remainder of this paper will map these concepts to the efficient running of a software organization. Specifically, it will provide a ready-made framework that the management can use to immediately start diagnosing what is happening.

**Chapter II**

# The Software Conveyor Belt

When a country wants to know how big its economy is, they use a metric called Gross Domestic Product, or GDP for short. It is the monetary value of what has been produced within the country over a period of time. It is calculated by measuring the value added to the input provided. For example, if a bakery produces bread and pastry worth $100 and spends $50 to procure flour, milk, eggs, and butter, it added $50 in value. Adding together the value added across all economic activities within the boundary of the country gives the value of GDP.

This idea of measuring the GDP can be applied to measuring the productivity of a software organization. If a team of 50 engineers ships features that bring in a revenue of $100,000 while incurring a cost of $50,000 they added value worth $50,000.

When a country wants to know how productive its economy is, it distributes the GDP over its population. This metric, where the value of the GDP is divided by the number of people living in the country, is called GDP per capita. This is each person's contribution to the value addition process. If the bakery is run by 5 people, the value added is $50, then the value addition per capita is $10.

In the case of the software organization, if 50 engineers add value worth $50,000, the value added per capita is $1000.

Finally, when a country wants to know how productive its economy is *really*, it measures the amount of work that has to go in to produce the value addition. That is, GDP per hour worked. In the case of the bakery, it would be how much work had to go in to produce $50 worth of value add. If the bakery kneaded their dough and whisked their eggs by hand, it would be reasonable to say that a lot of effort went in. If they instead used electronic tools to do this, they'd expend comparatively less effort. They'd produce less value per hour in the first case than they would in the latter. However, GDP-per-hour is hard to measure on the level of the economy and GDP-per-capita is used as a readily available metric.

The question that needs to be answered is how productive a software organization is *really*. While the GDP-per-hour metric may be difficult to calculate on the level of the economy, some thought can go into trying to quantify it on the organization level. To continue with the example of the bakery, say it takes 5 minutes to whisk the eggs, 15 minutes to knead the dough, and 30 minutes to put everything in a tray and bake a batch. We are able to measure the time taken by each step. In other words, we are able to measure the productivity of the bakers. If it took 20 minutes to knead the same dough, we could say that the process has gotten less efficient, and vice versa.

In the case of building software, the process starts with specifying what needs to be built. Whether it is a product requirements document, PRD, or a software requirements specification, SRS, there is something that codifies all this. From there, each individual component of the software is designed, developed, tested, released, and then monitored. The intuitive question is how much time it takes to finish each step.

The first step is to come up with a design. Coming up with a good design requires a certain level of research and prototyping. The amount of time and money it takes to acquire this knowledge, put it to use, and come up with a design can easily be quantified. In the context of a growing startup, ensuring that this phase runs efficiently means having, or starting to develop, processes that streamline each of the steps involved. For example, having a dedicated environment where a design can be prototyped to check how it would perform when integrated into the broader software. At this stage, the startup is beginning to grow large enough that it

needs a little more structure than it did in the early days where testing a design would simply mean hacking on the senior developer's laptop. The key here is realize what infrastructure needs to be built so that further designs can be easily prototyped.

The next step is development. The design is scoped, broken down into manageable chunks, and turned into raw code. In this step, there is a transfer of knowledge that happens because the design, which is perhaps a few diagrams and overarching goals, is now converted into individual tasks that will be completed by the developers who write the code. Therefore, effective communication is key to ensuring that this step proceeds smoothly. This can come in the form of written documents that detail the design or knowledge sharing sessions that are recorded for posterity. In the absence of a formal process of knowledge transfer, the chaos of growth, and the need to maintain shipping velocity, this knowledge of why the design came to be, how it can be evolved in the future, and what tradeoffs were made will be lost in the annals of time. At this stage of the startup, developing a historical archive of knowledge is just as important as shipping new features and bug fixes. This is also the precise stage where good development practices need to be enforced through automated means. For example, formatters and linters to ensure that the code looks uniform. The structure of the code, such that it lends itself to automated test suites, is also of vital importance. While some of these things, such as linting and formatting, can be automated, others require a shift in how code is written. Finally, having appropriate environments to deploy and run the code is just as important as they were in the design phase.

What follows next is testing. It is crucial for the startup's reputation that its software work with as few bugs as possible. Investing in writing comprehensive, automated tests that detect bugs early helps ensure that quality code is shipped every time. The suite of tests could be written by the developers themselves or by a dedicated quality assurance team. In either case, as was mentioned previously, structuring the code in a way that makes it easier to write tests is imperative. Like in the development phase, having environments where tests can be run is crucial for this stage of the process.

The tested code is then deployed. While in the initial days it may have been okay to deploy from the senior engineer's laptop using a script, at this stage of the startup, it is important to develop automated processes that allow multiple engineers to safely deploy their code as quickly as possible. The process of developing automated release pipelines applies to earlier stages of software development as well. In a nutshell, every time a developer wants to release code to production, it should be with automated pipelines.

Finally, the released code is monitored. Once the software is released, keeping tabs on its inner workings helps spot bugs and performance issues early. This requires coordination between the team that wrote the code and the one doing the monitoring. The metrics need to be injected in appropriate places to instrument the code so that the software can be monitored effectively.

Given how the process of creating software transitions from design to deployment, we can imagine the entire process like a production line on a conveyor belt at a factory floor. On the one end are requirements and on the other end is finished software. While a real conveyor belt is designed for mass production, software development is about developing novel systems. In a growing organization, however, patterns begin to emerge as the number of teams grows. For example, using a language and its ecoystem to develop microservices. This allows standardization of many aspects of process and accelerates the development process.

This conveyor belt analogy works well for iterative development, too, where the next iteration is a new requirement at the head of the conveyor belt. Productivity, then, is ensuring that the process goes from one stage to the next, as smoothly as possible, while maintaining high standards. Instead of measuring what gets done per hour, a better metric is what gets shipped per period of time like a couple of weeks, a month, or a quarter, depending on the complexity of task at hand. This becomes the metric to track.

It is now possible to answer the question of how productive a software organization is *really* by calculating the value added per engineer. If the revenue generated represents GDP, the intermediate costs like cloud costs and payroll represent the input. Subtracting the latter from the former gives us net revenue. Dividing this by the number of engineers results in the value added per engineer.

$$\text{Value-Add per Capita} = \frac{\begin{array}{c}\text{Gross Revenue} - \text{Cloud Infrastructure Bills} \\ -\text{Engineering Payroll}\end{array}}{\text{Aggregate Developer Headcount}}$$

Measuring the value added per engineer instead of raw activity metrics like pull requests, lines of code written, or tickets closed incentivizes the engineers to align themselves with what truly matters in a software startup: revenue. It requires that the teams ship lean, while still maintaining baseline quality standards, so that the software is used by the customers and starts generating revenue. This gives the management a metric they can track without putting the engineers under a microscope.

Measuring the value added per engineer has the advantage that it values highly intellectual, invisible work. Consider a senior engineer who spends weeks rearchitecting a critical piece of software that results in a reduction in cloud costs. It may be a few lines of code but they add a tremendous amount of value. Traditional metrics fail to capture this invisible work. In contrast, this invisible work will show up in the value added per engineer as the costs would have come down and consequently the value added would have gone up.

It also helps management answer the question of whether adding more headcount accounts for adding value per capita. If the expanded team results in increased revenue, the value added per capita goes up. However, if revenue stays flat, the denominator increases, and the value add goes down. In the latter case, it signals to the management that the software conveyor belt needs to be looked at closely because adding new engineers has not produced added revenue.

In conclusion, viewing the software development process as a conveyor belt helps maintain trust because inefficiencies happen because of slowness of processes than individual developers. In the same vein, calculating the value-add per engineer is a non-invasive metric that gives management perspective on how well the engineers are performing without resorting to micro measures such as calculating the tickets closed or lines of code written. The net result is clarity for management while maintaining its relationships with developers.

**Chapter III**

# The Human Telemetry

One of the many things that economists measure is happiness. This is done by surveying people and asking them questions that judge how happy they are. Organizations like Gallup and OECD (Organization for Economic Cooperation and Development) develop surveys which help understand the level of happiness in a given country. These surveys measure things like what a person felt about their life in the past, present, and the future. Aggregate metrics help understand the overall feelings of people. There are both supporters and opposers of measuring happiness with Albert Einstein famously saying that not everything that matters can be counted. However when running a startup it is helpful to know how the people are feeling.

The way startups gauge how the organization is doing is by conducting routine surveys. These surveys, sent by the human resources department, gauge things like how stressed the respondent is feeling or how likely they are to recommend the startup to someone they know. While questions like these are helpful and give the management a sense of what is happening, they fail to convey *why* the respondents are feeling that way. For example, if the stress levels are running high, is it because of intense workload or because of a lack of feeling of security?

As mentioned in the opening chapter, there are frameworks like DORA and SPACE that intend to provide a multidimensional view of the state of the startup. However, both of these leave the exact implementation of their methodologies to the ones conducting the surveys. For example, DORA mentions capturing "change lead time" which is the amount of time taken for code to go from being committed to version control to deployed in production. Accurately calculating this requires a high level of engineering effort which may be outside the scope of what the startup can afford to do at the moment given their constrained resources. As another example, SPACE advocates calculating activity metrics like the number of design documents written, the number of builds created, and so on. While these indicate the frequency of activity, they look too closely at the engineer and may erode trust.

For the management wanting to keep a tab on the pulse of the startup without incurring a lot of expense or eroding trust, an alternative methodology is required; one which gathers data at a system level without putting the individual developer under the microscope. Fortunately, the field of economics contains surveys that do just that. Specifically, the OECD Guidelines on Measuring Subjective Well-Being (2025 Update) demonstrates how to do this without being too invasive.

While the OECD survey focuses on subjective measures of individual happiness, a modification of their questions can be applied to get a sense of what is happening in the work place. These questions are split into two categories. The first category of questions focuses on how the engineers are feeling about their work as a whole. These elicit reflective judgement rather than current mood. The second category of questions focuses on specific aspects of their work. These pinpoint exact areas of satisfaction or dissatisfaction among the engineers.

The first category of questions is called "Work Evaluation" and presents three questions where the enginner answers them on a scale of 0 to 10. They are as follows.

Q1. Overall, how satisfied are you with work as a whole these days? (0-10)

This question elicits reflective judgement on how the engineer is feeling presently. Instead of being a reflection of short-term mood, the question asks the engineer to quantify what they

feel about work as an aggregate, taking into consideration all the factors they feel are important.

Q2. Overall, how satisfied are you with the work ahead of you? (0-10)

This question elicits reflective judgement on how the engineer is feeling about the future. It acts as a critical indicator of burnout and attrition. A drop in this metric signals to the management that the engineers are exhausted by the upcoming work or are unsatisfied about the processes they will have to continue working with.

Q3. Overall, how satisfied with your work were you in the past? (0-10)

This question elicits reflective judgement on how the engineer is feeling about the past. It acts as a critical indicator of decay or improvement in the processes. Comparing Q3 with Q1 provides a delta of what has improved or decayed.

The second category of questions is called "Domain Evaluation" and presents 8 questions where the engineer answers them on a scale of 0 to 10. They are as follows.

Q1. How satisfied are you with your standard of work? (0-10)

This question elicits reflective judgement on what the developer feels about the work they are producing: from designs to code. Engineers take immense pride in the quality of their work. A high score indicates they are satisfied with what they are producing and vice versa. A low score shows a systemic problem such as rushing the delivery of the software which is resulting in low quality work being produced. Put differently, the conveyor belt is moving too fast for the engineers to keep up.

Q2. How satisfied are you with what you're achieving at work? (0-10)

This question elicits reflective judgement on how meaningful the engineer thinks their tasks are. In other words, it helps identify whether the engineers feel they are making an impact or are stuck in a loop of busywork. A lot of busywork for top-talent engineers is indicative of systemic problems such as having to repeatedly wrangle broken environments. This takes away their focus from doing what truly produces value for the organization. Put differently, it is an indication of friction or prevalance of low-quality tasks.

Q3. How satisfied are you with your coworkers? (0-10)

This question elicits reflective judgement on the relationships among engineers. A high score indicates high trust whereas low score indicates low trust. Factors such as satisfaction of the engineers with the work of their colleagues, the trust they place in their judgements, and how they perceive the conduct of those who they work with are all subsumed in this question. Additionally, a high score indicates a high quality hiring pipeline that brings in vetted engineers whereas a low score indicates a broken pipeline.

Q4. How satisfied are you with how psychologically safe you feel? (0-10)

This question elicits reflective judgement on how at ease the engineers feel in the startup. A high score indicates a sense of safety whereas a low score indicates otherwise. Factors such as the trust the engineers place in their colleagues, the communcication of management with the engineers, and the day-to-day encounters with office politics all play a significant role in this

score. A lack of psychological safety is a leading indicator of attrition; engineers quit environments where they do not feel safe.

Q5. How satisfied are you with feeling part of a community? (0-10)

This question elicits reflective judgement on how connected the engineer feels to other engineers. As startups begin to scale and teams become loosely-coupled, a feeling of silo begins to emerge where the only people one talks to are the immediate colleagues they work with. A low score is indicative of siloing. Mitigating this requires fostering a sense of belonging through mediums that focus on shared interests such as choice of technology, level of seniority, and so on. A high score unlocks benefits such as cross-team knowledge sharing, avoiding re-inventing the wheel, and helping align the future of the technology stack within the startup.

Q6. How satisfied are you with your job security? (0-10)

This question elicits reflective judgement on how safe an engineer feels their job is. Factors such as recent restructuring, communications from management, the remaining runway of the startup, or product-market fits all contribute to this score. A low score is an indicator of anxiousness among the engineers and is a leading indicator of job security.

Q7. How satisfied are you with the amount of time you get to pursue other things at work? (0-10)

This question elicits reflective judgement on the availability of slack. Engineering is a dynamic discipline where things change rapidly and being able to stay abreast with them, either by upskilling on the current technologies being used, or by pursuing something completely outside of work helps maintain engineering satisfaction. In many instances, engineers have novel ideas that could help improve the current processes at work and having the time and resources to pursue these ideas helps unlock novelty.

Q8. How satisfied are you with your career trajectory? (0-10)

This question elicits reflective judgement on where the careers of the engineers are headed. Having a clear trajectory is an indicator of long-term retention of engineers. As such, management should spend effort in creating well-defined career tracks for those that work there. A low score is an indicator that the engineers are dissatisfied with their trajectory and could be either stagnant or unsure of where their tenure would take them.

In conclusion, these two sets of questions provide a holistic view of how the organization is doing. They elicit reflective judgement on various aspects of an engineer's work and provide valuable insights into how the organization as a whole is doing. Measured in aggregate, these are both lagging and leading indicators. With a simple proposition that "low score is bad, high score is good", it is possible to create a historical chart of these values so that the management can track the effect their policies and changes have over a period of time.

**Chapter IV**

# From Telemetry To Practice

Collecting a minimal set of non-invasive, diagnostic data points is simply the beginning. Its efficacy lies in the sophisticated interpretation. That is, putting the raw numbers into concrete practice. Knowing when and how to intervene to improve the processes based on the data that has been collected truly unlocks its value.

The value-add per capita metric can be calculated by pulling in data from accounting systems and cloud service providers. Calculating this metric immediately changes the way the engineering teams operate. It causes them to become more cross-functional as the revenue that is generated comes from more than releasing code to production. It comes from working closely with product managers and marketing teams to ensure that the software lands in the hands of paying customers. This immediately causes the engineering teams to ship lean and iterative code that moves the revenue needle.

The "Work Evaluation" questions help the management gauge how well their efforts to improve the processes have been received. Comparing how the engineers felt about their work in the past to how they are feeling today provides an indicator of whether the changes have worked. Additionally, it helps quantify how *well* these changes have worked.

The "Domain Evaluation" questions measure various aspects of an engineer's work. If the standard of work has dropped, the velocity of shipping code has increased to a point where engineering standards cannot be maintained. If the engineers are dissatisfied with their career trajectory, it could mean the career ladder needs to be more well-defined and more structure needs to be introduced to the promotion process. If engineers are dissatisfied with their coworkers, the hiring pipeline needs to be checked for brokenness.

All of these metrics, taken together, help the management see how the organization is faring. For example, if the revenue is flat but the "Work Evaluation" and "Domain Evaluation" scores tend towards the higher end, then perhaps product-market fit has not been achieved or marketing teams need to reconsider how they're reaching out to customers. Similarly, if the metrics tend towards the lower end, then the software conveyor belt needs to be assessed for inefficiencies. Perhaps the bottlenecks are in how the day-to-day engineering activities are being conducted.

For a busy executive, a dashboard with all of these metrics, presented both historically and currently, provides the tactical overview they need.

In conclusion, a careful interpretation of these metrics provides a holistic view of what is happening to the capital that has been allocated to the engineering teams. It shows how much value each headcount adds, allowing management to prudently hire their engineers. Comparing historical and current metrics gives a sense of what is happening in the organization while also pinpointing the exact symptoms that manifest themselves.

Table 1: Pipeline Diagnostics // Telemetry Intersections & Resolution Filters

| **Systemic Symptom** | **Operational Solution** |
|---|---|
| Gross revenue trajectories flatten while aggregate *Work* and *Domain* metrics register optimal high-end stability. | Absolve the engineering engine of failure. Route immediate operational audit to market-side monetization, product-market fit, and sales velocity. |
| "Work Evaluation" Q1 scores are lower than Q3 scores. | Look for decays in software development process. The growth of the organization has put pressure on the processes that sustained in the early days of the startup. |
| Engineers report low standard of work. | The software development process is rushed. Slow it down to let engineers ship quality work. |
| Engineers report low achievement at work. | Engineers are busy doing low-impact tasks like fighting with a broken dev environment instead of shipping what makes an impact. Create streamlined infrastructure that immediately unblocks the engineers. |
| Engineers report low sense of community. | Teams are siloed. Create technical guilds, shared architectures, and knowledge-sharing forums that enables cross-team collaboration. |