Modern Software Engineering

Aishwarya Singhal, January 2021

Introduction

Software Engineering has naturally evolved since the time the first programs were written. And so have the expectations of its consumers. Today's world *expects* everything to be digital. We use our smart phones to read news, to talk to our friends and family, and to perform most of our day-to-day chores. As consumers, we expect good websites, apps and technology enablement from all businesses. *(I am going to focus on websites and apps but the same principles can be applied to any software)*

This expectation has 3 constituents that define our happiness (or our perception of quality):

1. All features we have seen elsewhere must exist (feature parity with competition)
2. It must be easy and quick to use (customer centricity)
3. Everything must work without flaws (bug free software)

I have often defined bugs as "deviations in a software's behavior against stated or unstated expectations". (Even if no one said they expect a software to work in a certain way, if it does not, they will still be disappointed and will still call it a quality issue).

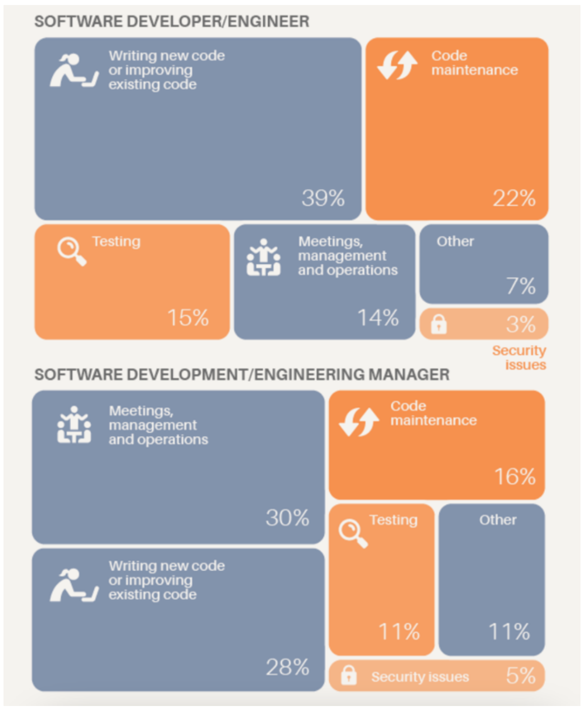
This in turn puts a lot of pressure on businesses and their IT teams (I intentionally draw a difference between the two. We will address it later). The businesses want to deliver to all facets of the customers' expectations, while managing the cost of delivering them. And the IT teams are flooded with requests, often overwhelmed with conflicting priorities coming from various stakeholders.

This makes software engineering much more complex than any other trade - a seemingly impossible scenario. It is only natural then that most IT teams do not deliver to the expectations of their business teams.

At the same time, my experience with various large scale IT teams showed a less than 60% time spent by developers on coding features. Even worse, many traditional organizations (businesses for which software is not the core product) have about 50% of team members than are in "overhead" roles - managers, coordinators, etc. - people that are not directly involved in writing the software on a daily basis. So, while there is an ever-increasing expectation of faster delivery, the actual effort spent on delivering the software is about 30-40%.

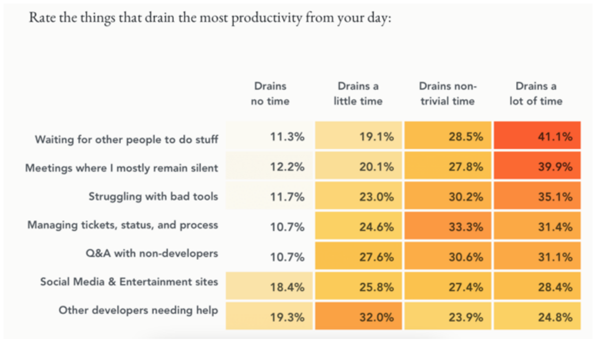
Let’s look at 2 exhibits I found on the internet.

Exhibit 1:



(*Source*: https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code/)

Exhibit 2:


Rate the things that drain the most productivity from your day:

	Drains no time	Drains a little time	Drains non-trivial time	Drains a lot of time
Waiting for other people to do stuff	11.3%	19.1%	28.5%	41.1%
Meetings where I mostly remain silent	12.2%	20.1%	27.8%	39.9%
Struggling with bad tools	11.7%	23.0%	30.2%	35.1%
Managing tickets, status, and process	10.7%	24.6%	33.3%	31.4%
Q&A with non-developers	10.7%	27.6%	30.6%	31.1%
Social Media & Entertainment sites	18.4%	25.8%	27.4%	28.4%
Other developers needing help	19.3%	32.0%	23.9%	24.8%

(*Source*: https://www.pluralsight.com/blog/teams/2017-software-developer-productivity-survey)

The numbers may be different per organization, but we know that the reality is not far off for most of them. So, it naturally begs a question - *can we fix this*? How do we maximize the software delivery, and cater to our customers' needs?

In this article, I intend to share my perspectives on the various tenets for this topic. We will explore the solution through 3 lenses:

- Defining a strategy for success
- Maximizing developer experience and writing high quality software
- Designing the organization

Defining a strategy for success

As leaders, we are often faced with challenges in balancing the needs of the business, and the constraints in delivering to those expectations. It is a complex problem, and one that requires multiple considerations.

Over the years, I have developed a list of 5 "principles" I found useful in defining a winning tech strategy.

1. **Speed trumps quality, but not always**

The speed to deliver a software, or a feature, the time to market, is extremely important. At the same time, it is important to focus on the quality. However, these two do not go hand-in-hand. Quality needs time, and that slows down delivery. And nobody likes something that lacks quality even if it is delivered ultra-quick.

The definition of *acceptable* quality changes based on the context. A throw-away software that enables a quick test of a business concept does not need to be perfect. However, a software that is related to hardware which can result in expensive losses due to bugs (like a space shuttle[1]), needs to have a much higher quality level. It is important to know the minimum acceptable levels for both speed and quality. How quickly do you need a feature, and how perfect does it have to be?

Usually, in a "Build vs Buy" discussion, "Buy" is more preferable. If you can find an opensource library that already solves a problem, it is better to use it instead of building from scratch. Similarly, a commercial off-the-shelf product may also provide a good foundation and jump start. However, ensure that sufficient due diligence (including a short proof of concept) has been done before adopting/ buying a software. There are a number of horror stories around off-the-shelf products. The marketing material always looks cooler than the actual fit of a software in your ecosystem.

> In general, an 80-20 rule helps. Ensuring that 80% of scope is delivered with >95% quality is much better than having 100% scope delivered with <80% quality, or only ~20% scope delivered with 100% quality.

It is far more important to be able to fix defects quickly, than to avoid them altogether. There will always be unforeseen issues once the software is released to consumers, but if you can fix that in minutes instead of days, nobody notices, and the impact is negligible.

An investment into technology - automated delivery (continuous delivery pipelines), monitoring, and processes that enable an on-demand deployment in minutes - will provide a much better risk management ability compared to any review processes that try to foresee and prevent risk.

2. **When in doubt, prioritize the customer**

In the B2C world, customer trumps everything else. Even if you are not in B2C, any software you produce needs to be optimized for the consumer. It is extremely important to define metrics and goals with the customers point of view. There are often conflicting priorities, and the engineers would always like to invest into ensuring a robust and maintainable system. As an engineer, I often find myself at the center of this conflict myself. However, as a principle, customer always takes a priority. It is *always* an unpleasant discussion, but a necessary one.

I read a quote from Steve Jobs somewhere:

> When you're a carpenter making a beautiful chest of drawers, you're not going to use a piece of plywood on the back, even though it faces the wall, and nobody will ever see it. You'll know it's there, so you're going to use a beautiful piece of wood on the back.

This is a cool quote, and I immensely respect Mr. Jobs, but perhaps this is something that **does not** apply to most modern software projects. For me, that plywood in the back may be perfectly ok as a way to get started. That does not mean that it should remain there forever. It should be replaced with beautiful wood as soon as possible. But we do not need to wait for the final quality until the software is released, as long as the chest of drawers is usable by the consumer.

Does that mean we let poor quality software to be developed? Absolutely not. Optimize for the customer and ensure that only the best quality is presented to them. At the same time, it is important to keep track of "technical debt" - compromises that have been made to urgently ship software to address a business or customer need. And it is important to have a *real* plan to fix that. Typically, a "technical budget" of 15-20% development capacity is a good way to ensure that the debt does not mount beyond unmanageable levels.

3. **Shipping software is far more important than perfecting it**

A few thoughts to keep in mind here

- Software sitting on a development or test machine is worthless until it is made available to the consumers
- The best way to perfect a software is to put it in front of customers and get feedback on it. There is no way one can

perfect a software without the customers providing inputs to it

- The longer you wait to release software, chances are that the quality will be lower. Counterintuitive? That's because the longer you wait, the needs of business are likely to evolve. Plus, it will be more complicated to merge all on-going changes being produced by the larger team, and it is more difficult to isolate and fix problems since there is too much change deployed at once

I remember working with a colleague who had previously worked in electronics industry - he was stunned that we could modify software and deploy "so quickly". In hardware world, they had to plan every change, implement the change on a breadboard, send the design to a factory for circuit printing, send the circuits over to the QA department, and work on the feedback. It took them weeks. That's not how software engineering works though, and it is important to recognize the difference. In today's world, if a software takes months or years to deliver, somebody's heart sinks.

There are various encouraging stories from leading technology companies. Amazon deploys every 11.7 seconds[2], and Google plans for 4 million builds a day[3].

How about the risk of errors due to frequent deployments? Risk management is often misunderstood. In my experience in software engineering, risk mitigation is far more effective than risk avoidance. As long as issues are immediately identified and quickly addressed on production. So, while all change managers will tell you otherwise, set an aspiration for your tech team to deploy multiple times a day, to production. OK - for a greenfield product, you need to first establish a minimum viable product (MVP) on production, before you have multiple deployments a day, but in that case, you only have a production environment once the MVP is ready. It is extremely important to have processes and technology that support multiple daily deployments.

I read somewhere: if you are not failing, you are not trying hard enough. Failure is not a problem, not being able to learn or come out of a failure is a problem.

Technically,

- Use Cloud for all deployments - ideally public cloud
- Automate everything - DevOps, Continuous Delivery, etc. Support zero-touch processes. Anything that requires a human interaction will slow you down
- Push for MVP mindset *across the board* and rationalize the scope for software delivery

Shipping software is probably second only to customer centricity in terms of a tech organization's priorities.

4. **Quality is directly proportional to the investment into talent and culture**

To start with, I am not talking about the financial investment only. I am also talking about time that you invest as a leader.

Now, of course, getting quality developers will cost a bit more than the cheapest available in the market. But you do not need the most expensive ones either. Having an all-star team does not guarantee quality. However, a team that sticks together, challenges each other, and believes in the goals of the organization goes a long way in establishing quality.

Similarly, the importance of culture cannot be overstated. My key considerations here:

- Hire quality developers and enable them for success. Let them take decisions. Collective brain power is always better than ivory towers
- Have a performance centric culture. Celebrate successes and capture learning from failures. However, ensure that people are not scapegoated for failures. The only failures that need to be discussed are where people were

comfortable with their status-quo and failed to try or innovate

- Ensure alignment of common language and goals across the organization. As long as there is a separate "business" and "IT team", quality will suffer. Ensure that the same goals are used for both, and that they are working as collaborators. ***Software needs to be business led, and not IT led*** (although the tech team needs to have a sufficient degree of freedom to bring in tech innovation). Encourage everyone to think of the customer. It is not just the designer's problem, or the customer service department's. Spend time with the teams, so they feel connected
- Invest into the best tooling for the developers. High quality tooling improves productivity, encourages creativity and innovation, and improves people retention. E.g., buying good laptops for developers is a one-time cost, and not a great cost, but significantly improves the quality of their output. Good tooling can also improve collaboration and cut down on unnecessary meetings, which further improves the productivity
- Ensure that *everyone* is learning from external community (outside of your company) via meet ups, conferences, or talks delivered by external experts. This needs to happen frequently, and the experts need to be real experts, even if they do not speak the local language

> Getting quality delivered to customers is hard and it will only happen when the whole organization collaborates, instead of throwing it over the wall to the "IT team".

5. **Be bold: there is no replacement for testing and learning**

> The road to wisdom? — Well, it's plain
> and simple to express:
>
> Err
> and err
> and err again
> but less
> and less
> and less.
> - Piet Hein

There is no shortcut to testing.

Before the teams start building a product, test the business case. Conduct user tests on cheap prototypes. Not every fancy idea is worth developing, and what may work for another company in another set up, may not always work for you. As a leader, you can (and should) help the teams rationalize their requirements.

Once built, measure everything, and capture as much customer feedback as possible. Invest into analytics and capture every customer interaction. Analyze the data for any trends, and feed that back to the technical teams' "backlog" to be prioritized and implemented.

Also, that means that approaches like Mechanical Turk[4] which involves setting up "fake" solutions until "proper" solutions are available, can be fantastic in getting customer insights.

The cycle should be: Build -> Measure -> Learn -> Repeat[5]. Shorter this cycle, the better it is.

However, a balance is important as always - avoid rabbit holes and know when to pivot. A VC-like mindset is often helpful. So be a coach for the team, encourage testing, but also encourage learning from others and to let go when tests *consistently* reveal negative results.

At the same time, encourage the teams to be bold and bring in innovation from around the world, and not just constrain themselves to a specific sector. Every idea is worth testing.

In the end, there is no silver bullet solution, and you will need to review all of these in the context of your organizations. But I certainly hope that these may warrant a discussion within your leadership circles and help define a strategy that works for you.

Maximizing developer experience and writing high quality software

Is the practice of developing software a science (Computer Science), an engineering (software engineering), or an art (software craftsmanship)? When I was in university, we always viewed software as science. We experimented, we learned, and we treated it as mathematics - driven by pure logic. When I started working, it became more of an engineering - applying known techniques, searching for ways how others have solved a problem before, and looking for efficiencies. In recent years, I was introduced to the idea of it being a craft - i.e., focus on quality, and believe in the fact that it can always be improved. I can't say I fully practice craftsmanship; however, I have moved from engineering more towards it. In my personal view, most projects unfortunately do not quite allow for (or warrant) the time needed for the craft. In any case, we can *always* do a few things:

1. Ensure a certain level of quality the first time we publish the software through automated checks and thorough code reviews, but avoid over-engineering
2. Make time for refactoring
3. Work smart, not hard - get as many open-source libraries as possible to solve your problems, and only write code for things that are truly specific to your problem and cannot be found on the net

Based on these 3 ideas, I have a few practical tips.

1. **Build on best-in-class programming techniques**

My favorite here is the UNIX philosophy[6] that was published in 1978 (yes, over 40 years ago!):

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
- Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
- Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Why do I love these? These have stayed solid (as has UNIX) over the past 40 years. I derive my coding principles from these, and the following are my most commonly used ones at the moment

- Write short methods that do one thing only and do it well. This in turn helps to keep a low cyclomatic complexity as well as a smaller number of lines of code per method. I love the Unix pipes and filters, and if you can build that idea into your methods (e.g., using Strategy pattern), a fantastic code quality emerges
- Use microservices[7] (small pieces of functionality that are independently deployed) where possible

- Go minimalistic in your design of interfaces, following the YAGNI[8] and Convention over configuration[9] principles. Try to follow the DRY[10] principle as much as possible (without making the code too unreadable)
- Insist on modularity so that pieces of code can be thrown away when not needed. Caution: avoid over-engineering. This is not the most important aspect if you are following the other principles
- Refactor, refactor, refactor: Do not shy away from refactoring. The principle is, *whenever you look at a piece of code, aspire to leave it in a better state than you found it in*

A technique I find useful in writing modular code is:

Every time you feel the need to write a comment in the code, see if you can make a new method/ service. Comments usually indicate that the code is doing more than what can be easily understood.

You can use any programming language, and any style - my personal favorite at the moment is Functional programming in whichever language I use, because it helps me implement the above-mentioned goals easily.

A technique not mentioned here and one I am a big fan of is Event Driven Architecture[11], (or alternatively, Reactive Programming). It helps reduce dependencies and provides an easier way to guarantee performance and reliability of a system.

2. **Align on quality goals and then automate them**

I have seen situations where the team discussed at length, and kept discussing, the choice of technology. I have also seen similar debates around quality. The only way to avoid an endless debate is to propose and align a set of technologies and quality measures *democratically* with the team, and then adhere to them for at least a few months. And the best way for that to happen is to automate the agreed principles.

Do not define a quality goal that cannot be (at least partially) automated, because it is unlikely that it will be implemented.

- Be real about your Definition of Done (or equivalent) and hold your team accountable to it during code reviews
- Timebox all decisions
- Try to leverage industry standards where possible - e.g., Airbnb style for JavaScript linting is often used by teams, or like back in the day Sun's Java conventions were pretty standard guideline for Java code
- Call a meeting at the start of the project and agree on quality goals (and publish them)
- Anybody joining the team afterwards can give suggestions on these goals, but they should only be accepted if they do not disrupt the rest of the team. Alternatively, they can be accepted in the next review of the quality goals (after at least 6-8 weeks)

As I mentioned earlier, bugs are any deviation from a user's expectations. That includes functional defects, and also performance, usability, reliability, etc. Ensure that your quality goals take a complete view.

Typical techniques like TDD, Code Reviews, Code Style checks (static code analysis), etc. are usually good measures. When writing automated tests, it is more important to have real quality tests than writing for the sake of getting a 100% test coverage (e.g., you must get a 100% coverage on code containing logic, and it is ok to skip tests for simple Value Objects).

Some aspects can be only partially tested - e.g., in case of web accessibility, or security, a manual review may still be required. However, there are many tools available to get you an 80% correct view (if not more) and I would highly encourage using them.

Similarly, take your code reviews seriously. GitHub and similar tools simplify the review process significantly and can integrate all feedback that automated tools can generate to help you review code.

Technology evolves fast, and I would always recommend checking for the best ways to achieve automation of quality goals before starting any project, and every couple of months even after starting a project.

A manual review by the product owner or quality engineer may still be required, but by the time it goes to them, all other checks would have ensured a decent quality level.

3. **Be truly agile: ship the software as soon as possible**

As I said in the earlier, shipping software is far more important than perfecting it. As long as the code meets all quality goals, it should be good to deploy.

I always reflect back on my days in the school, when I first started to code. This was what SDLC looked like to me then:

1. Get a problem (requirements) from the teacher (product owner/ user)
2. Implement them on my computer
3. Copy the working code into a floppy disk (deploy)
4. Show it to the teacher

It did not take me weeks or months to do that. It was often done from one day to the next, and in some cases even during a class.

Even when we had a project where multiple teammates were working on it, the cycle only had one more step between 3 and 4: Integrate your code with a friend on their computer (aka production set up)

There was no 3- or 5-environment set up, no change management, no design approvals, etc.

We have made software development overly complex over the years, and it is important to simplify it. The longer you take to ship software, the worse quality you can expect.

Now of course, you need to have processes and checks to ensure quality of delivery. However, as long as you have defined sound quality goals and the code meets them all, your code should be good to ship. Put it in front of the customer and address any learnings that come out of that. If you can fix issues quickly, it is perfectly ok to have a few bugs that pop up once you deploy.

Some ideas:

- Use feature branches and feature flags for software development, and have a process to clean up stale feature flags once a feature has been stabilized
- Ideally, you should push your code to production at least once a day. In the worst case (for complex and large problems), push it within a week. For sub-projects (like a redesign) that take longer, create a pipeline to deploy the feature branch on the test environment for that sub-project – that's your production environment for the sub-project. In no case, keep the feature branch alive for more than a week
- Fully automated deployments: Use continuous delivery pipelines and allow developers to build their own infrastructure through scripts/ bots (infrastructure as code). Achieve a full automation on deployment, ideally including the production environment. In highly controlled settings, implement a fully automatic deployment pipeline at least until pre-production/ staging environment, and then a 1-click deployment for production
- Ensure sufficient monitoring and logging in the code to observe and learn from user behavior. That will ensure a much higher level of quality than what can be predicted during the development phase and is absolutely needed for a continuous

delivery system. CNCF[12] is a great place to start for such topics.

- Optimize the delivery pipeline to take less than 30 minutes (faster is better) including test execution. This will ensure that developers get feedback on broken builds and issues ASAP and are able to quickly fix the issues on production.

One last tip here - be honest to yourself. Every time you have to do a less than perfect job, note down a technical debt item in your product backlog so it is tracked and never forgotten.

4. **Reduce the number of meetings you attend**

One of the main time-drains for developers is the number of meetings that happen. Avoid them. Put a limit of a total of 30 minutes per day for meetings that need more than 5 people (e.g., the morning standup) for at least 4 days a week. The exception will be some days when you have an architecture/ design session with whole team, planning meeting, or a retrospective, etc. These longer meetings should be on the fifth day of the week.

Try to move as much communication online as possible. Use tools like Slack to have effective integrations with various tools and have chats with your team. An online discussion has various advantages - you only dedicate time that you absolutely have to. Also, it helps any other team member to pitch in or learn if they see value in the topic - that makes it much more productive.

It is vital to understand the true meaning of agile and I recommend re-reading the Manifesto[13] and listening to the talk Agile is Dead[14] every few months. More often than not, teams claim to work in agile manner but still have numerous complex processes and constraints built around them. Whenever you get an invitation for a meeting, ask yourself - can I avoid this meeting? Try to skip as many meetings as possible. At the same time, pair programming sessions can be awesome. Take a pragmatic view and do those whenever it makes sense.

One of the reasons for meetings is dependencies and integrations. Can you reduce them? Try to design your coding responsibilities so you can own end-to-end slices and have minimum dependencies on other teams/ team members. Use interface contracts[15] along with techniques like Mechanical Turk, stubs and mocks, to be able to independently develop your code. When done well, this can be done completely independently, and results in significantly reducing integration efforts.

Lastly, when I talk about meetings, I am excluding the ones that help you learn (e.g., conferences, meet ups, knowledge exchange sessions). Try to carve out time for them so you do not disrupt your productivity too much, and yet have reasonable time available to learn and share knowledge.

As a thumb rule, you should be able to get 6-7 hours a day for focused coding.

5. **Leverage and contribute to Open-Source Software, and Internal Open-Source**

A key aspect to optimizing your time *and* improving the quality of your code is to leverage open-source libraries as much as possible. Every time you have a problem to solve, check if there is a library that already does that. Ask your team. There is a library for most of the commonly encountered problems - somebody somewhere solved it, stabilized it, and published it. Beware that there are also a number of bad libraries out there, so make sure that a) there is sufficient community behind it, and b) you have tested and seen it working.

Open-Source is awesome because people contribute to it. See what you can publish too. If you have solved a generic problem, publish a sanitized library (check for your organization's policy first). It helps the community of developers, but it also builds a brand for you and your company and attracts good developers to work with you.

Similarly, see if you can build an "internal open-source". If a colleague needs to re-use a piece of code, or if you are re-using code written by someone else, see if it can be a library to be shared internally (or if it generic enough, externally too). Do not greedily create libraries, but instead let that be done on demand. This ensures that a good ecosystem exists for all software in your organization, and everybody benefits from your learnings.

At the same time, allow anyone in the organization to submit a pull request, or make changes to the library and help evolve it. That's the true nature of open-source software and helps with its adoption. Failing this, it just becomes a framework component that will always be your responsibility to maintain and fix, and will also see skepticism from your colleagues on its adoption.

Finally, find time to learn. Time spent on learning yields exponential results in your productivity (and happiness). Keep measuring quality of your code (through different tools), and you will master it.

Happy coding!

Designing the organization

Typical IT organizations have evolved into having multiple layers of managers. Some of that is because organizations try to reduce their risk by having more managers reviewing the work being done. Some is because the growth model only supports growth as managers, and hence everybody grows into a managerial role sooner or later, leading to a pyramid of people that are primarily in supervisory roles. Many organizations have as much as 50% staff in supervisory/ managerial roles. Simply speaking, only 50% of the staff is involved in actual production of software.

Basic economics implies that typical overheads (or SG&A) in an organization should be about 20-25%. Shouldn't the same logic apply to IT teams too?

Another aspect to consider here is that complex organization structures lead to *a lot of* meetings that waste productive time.

At the same time, there is the question of quality being delivered and the trust between different teams. Often, we see a "handover" mindset in most teams - they deliver their part, and then any issues found are to be fixed by the team that comes next in the chain. More often than not, the end-user's perspective is ignored and forgotten, and teams focus more on covering their backs than on doing the right thing for the user.

Let's look at all these aspects through various enabling mechanisms.

1. **Aligned goals and metrics**

A key aspect of ensuring quality in deliverables is that there is a common definition of quality across the organization. Most teams fail to recognize this, and we see different metrics being used by them. So, while a sales team might be tracking revenue, or customer service team might use Average Handling Time (AHT), the IT team enabling them might still be measuring the number of code releases, or bugs. Now clearly, there is much more than goes into enabling high revenue or low AHT than the software, and there are a lot of IT specific aspects developers need to care for, but that does not mean that the software developers do not have a view on these business metrics.

It is vital that everybody uses one language and common metrics across the organization. My most impactful stories have been from situations when my teams took the end-user view and partnered with the stakeholders to ensure that the end result was beautiful. Magic happens when developers and business teams collaborate on achieving common goals.

One simple example - we had a feature request to enable printing of VAT invoices for customers, and the developer on my team had already implemented it. However, he did not look happy. I walked up to him to find out why, and I saw him with a printout of an invoice and an envelope. He was upset that the printed customer's address did not fit in the center of the address cut-out on the envelope. He did not have to do that test, but he went out of his way, fetched an envelope, printed and folded the invoice, and checked if it will work.

On the other hand, I was in a team for a large company whose main business was through online sales. Their website crashed, and it had been down for 2-3 days. We were parachuted in as external experts to rescue and fix. At 5 pm, the developers picked their bags and were leaving. We asked the lead developer if he could help debug the issue and he refused - it is the job of the support team and they need to manage it. Now it *was* late, so I get his point-of-view. However, in such a situation, I would expect an all-hands-on-the-deck type mindset.

The disconnect between software developers and business goals is sometimes shocking.

Most successful set ups are where all software teams have a business leader who is committed to enabling success and is not just a stakeholder. These business leaders also have sufficient say in the system, typically a direct line to company's leadership. And in such cases every software team is directly responsible for their impact on the business metrics.

Sure, there will be IT specific metrics that the developers need to track, but they also need to have a keen view on the business goals.

I recommend having large screen monitors (that show both business and IT metrics) next to where the developers sit, and that the teams include the business metrics in their performance reports (demos/ sprint reviews, …) at least once a month.

However, you do not need to over-engineer this. You do not need to track business value or cost per feature. A meta level view is just fine. The goal here is to establish better quality via ownership and awareness, and not to bring in an accounting overhead.

2. **Product and platform, not project teams**

Many organizations work in an outsourcing model even with their internal IT teams. The business team creates a project, gives it to the IT team, and then the IT team has the responsibility to deliver. As expected, this helps optimize the costs (*maybe*) but erodes quality and trust.

The issue here is that most organizations have one model for day-to-day functioning and for mentoring and reporting. This does not have to be.

It is important that organizations drop the notion of projects and move towards products. Now "product" has a specific connotation in most organizations - however, we are not talking about the product that you sell to your customer. We are talking about the *"software product"* that will enable that sale. Although you may sometimes align software product teams with actual products that will be sold to the customer.

The difference between a product and a project is that the latter has an end date. It is important that there are product teams that take an end-to-end view on a product, and not a tactical view on enabling a feature/ few features. This enables an improved view on quality and ownership in the teams. This also enables an easier way to align KPIs/ OKRs with the business teams.

An easy way to create product teams hence is to follow the business metrics and their responsible business leaders. So, sales may warrant a developer team, customer service might warrant another, and logistics might need yet another. All of them may warrant multiple teams, depending on the number of metrics and business leaders.

Another interesting tactic is to allow each business area to have a budget for software development and let them allocate it to each product team based on the latter's performance in their QBR presentations. This drives collaboration between the business sponsors and the product teams.

When you have multiple product teams for a common business area (e.g., sales), you just need all product owners to collaborate with the same business responsible person.

3. **Your organization structure does not need to reflect your IT architecture or your vendor landscape**

Many IT teams adopt a n-tier architecture, which is composed of different layers. Many of them model their organizations to align with the architecture too - there is a frontend team, a middleware team, a backend team, etc. This leads to a large number of dependencies (and bottlenecks) across teams, and also a lack of end-to-end ownership.

In my experience, the most effective model is when organization structure does not replicate the IT architecture. In such cases, there are product teams with *end-to-end* responsibilities, and platform teams that enable the product teams with *common* tools and frameworks.

The platform teams, or as we alternatively call them - IT-for-IT, are deeply technical teams that develop tools and frameworks. Think of these teams as R&D or enabling teams, whose customers are the product teams, and whose primary responsibility is to bring in efficiency and innovation. These are extremely important, and the product owners for these teams need to directly report into the IT leaders.

Although we call these platform teams, they should not be centered around specific technical tools, e.g., a Salesforce team, or a SAP team. Salesforce experts, or SAP experts, should be embedded in the right product team.

In some cases, the work required is too much to be handled within one "full stack" team. In such cases, there are 2 options, viz., a) take thinner slices of work so a lean team with end-to-end responsibility can still work, or b) divide the teams based on 1-2 layers such that they still have a business significance (e.g., one team does everything until API-enablement, and other prepares frontend and integrates the APIs). The second option is less preferred, and as much as possible, end-to-end ownership should be ensured.

4. **More pigs than chicken**[16]

You need more people that have their skin in the game than those that are just supervisors or advisors. My typical assessment works on the following lines:

- Anybody who is not actively building or maintaining a product, nor takes an active part in defining the requirements, is an overhead. This includes all advisory roles - security, privacy, architecture, coaches, etc.
- Anybody spending more that 50% of their time in meetings is an overhead
- The total number of **overhead roles should be less than 25% of the total organization**. So, if the IT team is 100 people, at least 75 of them must be actively building the product

A simple way to start is to de-layer the organization. A product owner should have a direct reporting to the business leader responsible for that area, and all developers work directly with the product owner and the tech lead, and all tech leads work directly with the IT leader (CIO/ CTO/ VP/ ...). Cut down on all other managerial layers, and clearly define roles and responsibilities for every role.

Ensure that the Product Owner comes from business team's perspective and is responsible for writing clear requirements, and for verifying the implementation, and the Tech Lead is a senior developer with >80% time dedicated to coding, and remaining time for mentoring the team.

Automate all non-value adding tasks, and simplify what cannot be automated, e.g., coordinator functions, where someone is only responsible to raise a ticket or act as a SPOC for communication. Another example is replacement of manual QA work with automated tests as much as possible.

Similarly, as an example, all advisory roles could be staffed on product teams as needed and would be expected to have an acceptable utilization rate.

Typically, such an exercise frees up between 15-20% of capacity that can then be reallocated to value adding roles. The freed-up people are also very talented people in wrong roles, and normally >95% of them can be reallocated (and will be interested) for further value creation. Some might need a bit of training; and investing into them brings out magic. Congratulations, you just created a significant productivity boost (through saving and reallocating).

At the same time, as a word of caution, do not go overboard with this idea. Many of the advisory teams are often understaffed and underappreciated. In some cases, having SPOCs helps product owners and business leaders to maintain their sanity, especially when it comes to managing vendor relationships. You may still need some manual QA. Similarly, all organizations do require managers, so trying to move towards a near zero managerial capacity will be an absolute disaster. While it is important to chart out an ideal picture, it is also important to then apply a prudent lens and ensure that the model will work in your context.

A study at Google[17] indicated that the most effective teams are the ones where team members feel psychological safety and have structure and clarity. I recommend keeping this as the underlying though when designing the organization.

5. **2 pizza box teams**

This concept came from Amazon and is almost an industry standard now. The idea is that the team is small enough to have a healthy collaboration and can work together as a SWAT team to deliver towards a common goal. My recipe for typical teams is: 1 Product Owner, 1 Designer, 1 Tech Lead, 4-5 Developers, 1 QA, and 1 Advisor. The designer and advisor role may be fulfilled by different people at different points in time of a product release, based on the need. E.g., there may be a UI designer at 50% and UX designer at 50%, or 50% of architect, 20% of security, and 30% of Subject Matter Experts/ coaches. Some of these may be shared across different teams. So, there are 7-8 dedicated team members, and 2 that are floating. The reason why I would count the floating also into the team is because these need to be in the stand-ups and need to be accountable for the quality of delivery (i.e., they need to be pigs, and not chicken).

In special cases, depending on the complexity and (lack of) maturity of the organization, some teams may also have a Business Analyst/ Junior Product Owner, someone that helps the product owner by taking up some of their responsibilities.

6. **Functional vs Reporting structures**

One important clarification needs to be made here. Everything above talks about how the teams should operate, and not where they should report. The IT team members should continue to report into the IT leaders, so that their career growth, learning and mentoring can be shaped by leaders that understand the field.

The product teams should have a dotted line reporting to the business leaders, and the feedback on their performance should be evaluated based on their performance in that context.

Another thing to note here is that **this does not mean that the IT leaders report into their business counterparts.** Both IT and business leaders need to have a top-level reporting into the company leadership. This is necessary to ensure that the organization does not always prioritize tactical goals over technical excellence and innovation.

This model ensures that the business leaders do not need to worry about the mentorship of technical teams, and the teams get guidance and support from leaders that understand the space. At the same time, the technical teams are focused on generating business value for the organization.

7. **Chapters, or communities of practice**

A final missing piece here is knowledge sharing. It is important that teams share their work for 3 reasons:

- It enables consistency of implementation across the organization. People have an ability to challenge each other every time they spot an inconsistency. This in turn helps with cost optimizations via prevention of fragmentation and avoidance of duplicate costs
- It enables learning within a community of similarly skilled colleagues
- It helps identify training needs for specific skills

Spotify has Guilds and Chapters[18]; many other organizations have communities of practice[19]. It is vital to encourage creation of similar *virtual* structures and ensure that they are exchanging knowledge on a regular basis. So, the community needs to appoint a leader, and that leader should regularly share their observations with the IT leaders. Note that this is not a dedicated role, but an additional responsibility for an existing team member.

This has an interesting side-effect: it enables a different growth model in IT compared to traditional ones. Developers *can remain* developers and still grow (in responsibilities and financial sense) without taking up managerial roles.

As always, there is not just one answer for organization structures. Different models work for different set ups, and it is important to understand the context you operate in, and what works in that context. Similarly, the size of an organization can play an important role in defining the feasibility of some of these measures. What works for a 50-member team may not work for a 5000-member organization. Finally, culture and team maturity play an important part in defining the model.

At the same time, the principles remain broadly the same, and as long as one can define an execution model that works in their context, it will enable a significant productivity and quality boost in the output.

So how do we solve for large organizations? Well, for one, there are a number of standard frameworks and methodologies. I hear SAFe[20] is the most famous. I am personally uncomfortable with any “one-size-fits-all” solutions, so I would recommend evaluating the options based on your context and devising an execution mechanic that works for your organization.

Finally, at the heart of all these tips is the intent to simplify (reduce complexity). Anything that increases overheads or complexity *in the long term* must be challenged and re-evaluated for fit in your context.

References

1 https://www.bugsnag.com/blog/bug-day-ariane-5-disaster
2 https://techbeacon.com/devops/10-companies-killing-it-devops
3 https://thenewstack.io/google-reveals-the-secrets-of-devops/
4 https://en.wikipedia.org/wiki/The_Turk
5 http://www.startuplessonslearned.com/2010/09/good-enough-never-is-or-is-it.html)
6 https://en.wikipedia.org/wiki/Unix_philosophy
7 https://martinfowler.com/articles/microservices.html
8 https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it
9 https://en.wikipedia.org/wiki/Convention_over_configuration
10 https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
11 https://en.wikipedia.org/wiki/Event-driven_architecture
12 https://www.cncf.io/projects/
13 https://agilemanifesto.org/
14 https://www.youtube.com/watch?v=a-BOSpxYJ9M
15 https://saucelabs.com/blog/intro-to-contract-testing-getting-started-with-postman#:~:text=Contract%20testing%20is%20a%20way,the%20data%20the%20client%20needs
16 https://en.wikipedia.org/wiki/The_Chicken_and_the_Pig
17 https://rework.withgoogle.com/print/guides/5721312655835136/
18 https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf
19 https://en.wikipedia.org/wiki/Community_of_practice
20 https://www.scaledagileframework.com/

This article is a compilation of blogs published on Aishwarya's website at http://asinghal.github.io/ and contains his personal opinions. The content is shared for the sole purpose of encouraging leaders in software engineering field to debate and design their own solutions based on their contexts.