#superagile

An introductory guide to superagility
and how to succeed at it.

Brought to you by

concise

Table of Contents

Why #superagile?

#superagile is designed in support of building scalable teams and software. It has a growth mindset and makes it easy to respond to new business opportunities in a quickly changing world.

It is not a process; it is a framework, approach, and a type of culture every startup benefits from. It combines top-notch technology with agile principles and adds a growth hacking mindset. It brings business, marketing, and development closer together to reduce expensive communication faults while spreading ownership feeling across all team members.

In the end, **it helps save time and money**. It makes innovation cheaper by paying attention to the right thing at the right time.

It is an honest and open approach to successful digital product building.

Read more:
https://concise.ee/superagile
https://concise.ee/blog/how-does-superagile-differ-from-agile
https://concise.ee/blog/what-the-hell-is-superagile

FOLLOW US ON SOCIALS

How #superagile was born

#superagile was born on the 10th of January, 2019. It was created by Aive Uus, Mikk Soone and Andrei Zhuk. For over half a year Aive, Mikk and Andrei had been involved in looking deeper into Concise's way of working and vision for the future. Agile principles were already widely in use. It was not just about having a few Scrum practices and saying it is agile. **It was more about understanding the real meaning behind it and taking action.**

One Conciser recently gave feedback saying: "Everyone speaks about agile, but I have never before seen it working. It seems like a mystery how many companies find it challenging, but here it feels so simple and natural."

In addition to agile principles, this approach involves more specifics for ambitious digital product builders. The word **#superagile** was born a few days before the Concise Growth Team (Aive, Andrei, and Mikk) flew to Helsinki to do their first sales pitch. Why was this word chosen? **While testing it out, this was the word that got the most emotional reactions.** People either loved it or hated it. But nobody stayed neutral. Some said agile is just a big buzzword and some enthusiastically got interested in whether it takes agile one step further. How did the pitch go? Horribly. As first pitches usually do. Nevertheless, something great came out of it and thus, **#superagile was born**.

A few months later, the **#superagile score tool** was also born and has been an internal teamwork practice at Concise ever since.

In this book, we take a detailed look into the 13 elements of that tool.

Our app "Superagile" is out! It will help you to conduct the workshop yourself for your team! Get it from the App Store or Google Play.

People behind the book

Aive Uus

Passionate about growth and teamwork. She has been leading software teams for close to 20 years in a creative way and enjoys building and supporting highly effective product teams.

Mikk Soone

Passionate developer who builds scalable software products. Minimal-time-to-market? Quick business value? Yes, always.

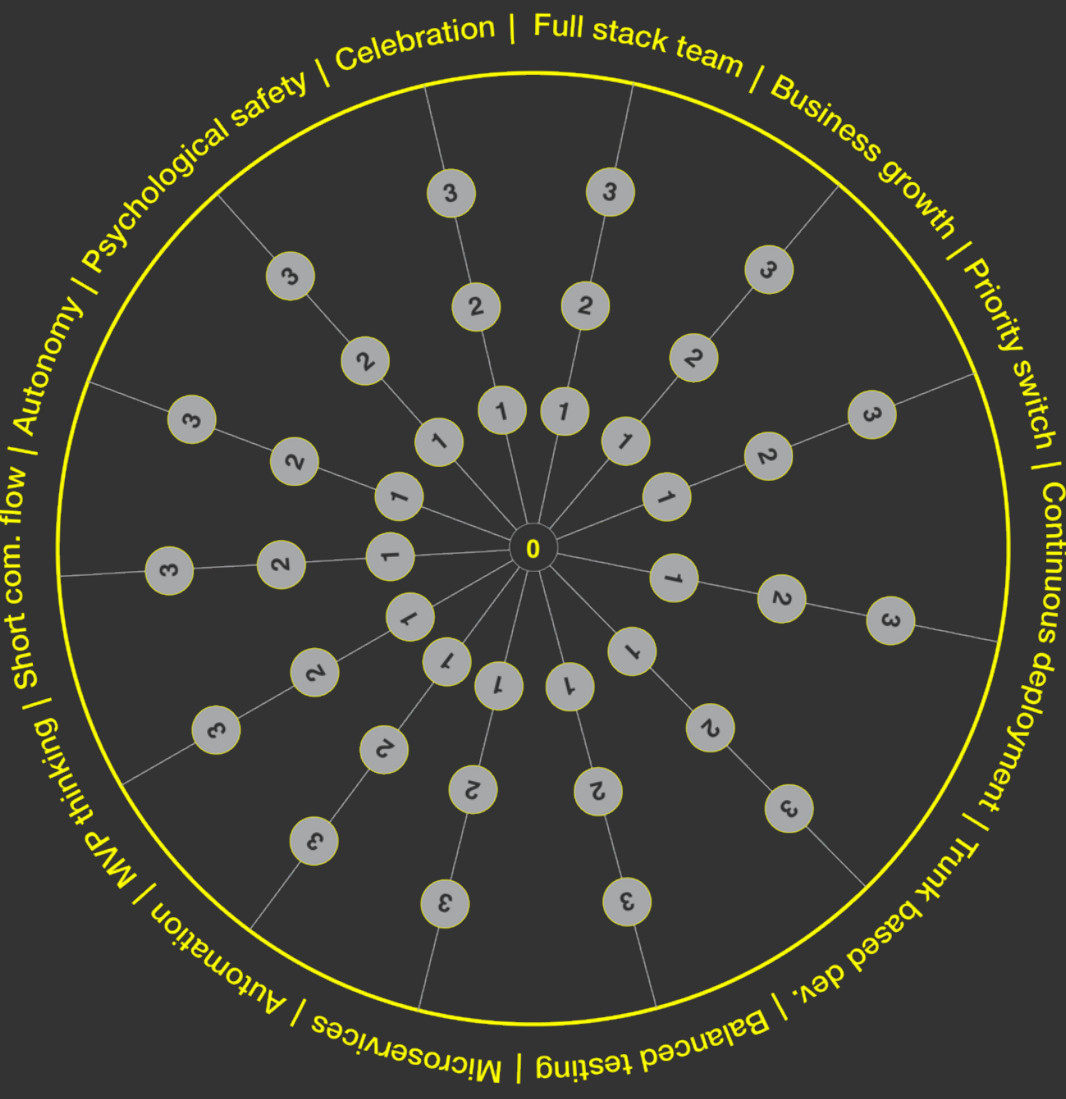
Indrek Ots

Software engineer who's interested in building resilient systems and fostering a culture of learning in teams.

Loore Martma

Combining the knowledge from a creative mindset with health-behavior and psychological well-being. Supporting teams in their communication skills to raise awareness of a mindful workplace.

Also, big thanks to ***Amanda Lucas*** for copywriting and editing, and ***Lisett Kruusimäe*** for design and creating an actual book out of it!

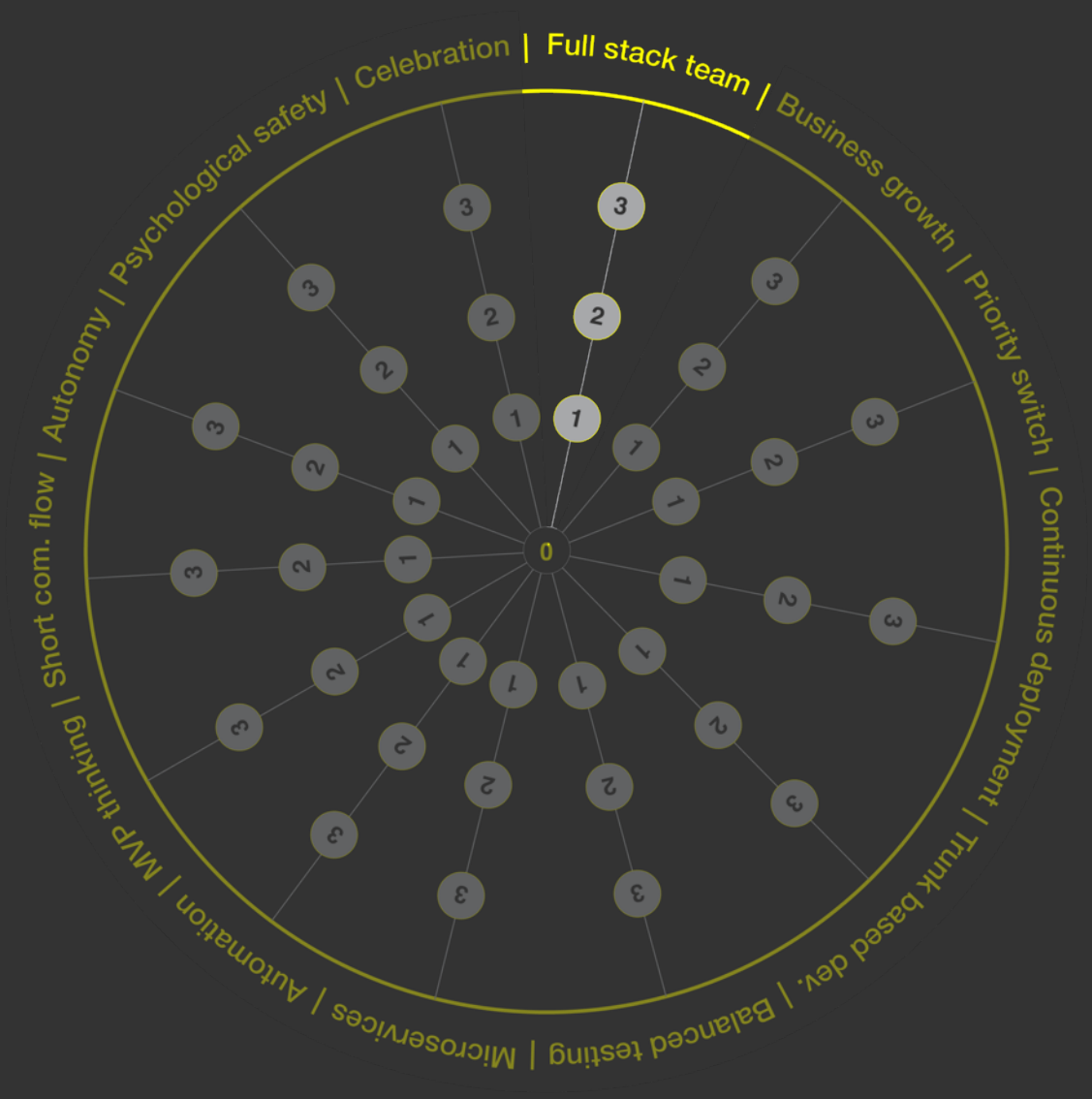


This is our #superagile wheel
which you can personalize for your team or use for auditing.

Contact us to find out more about how to use it
or download our app from App Store or Google Play

CHAPTER ONE

FULL STACK TEAM

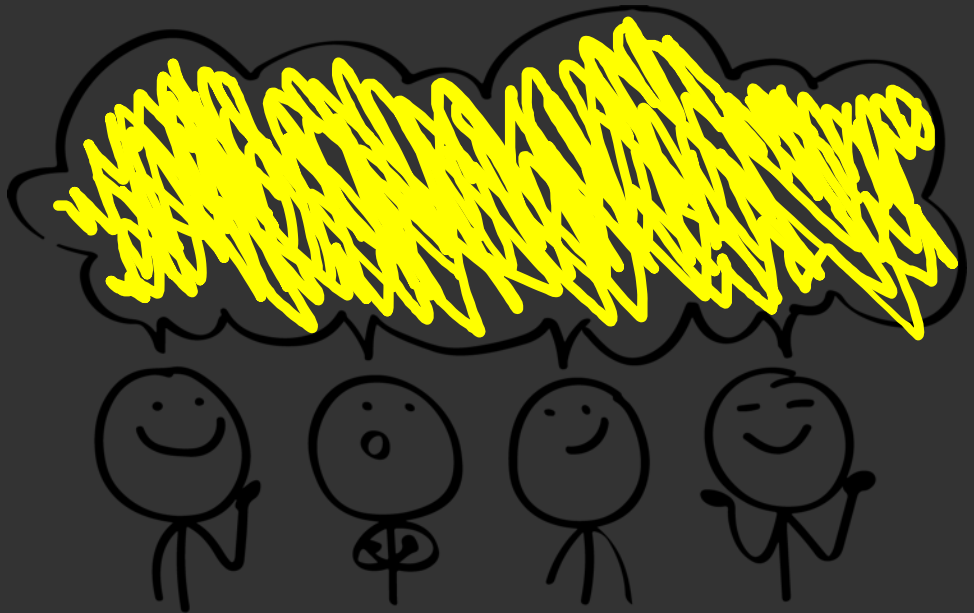


It all starts with the team. 2+2=10 is a well-known term describing good teamwork or in other words:

“The whole is greater than the sum of its parts.”

Have you ever thought about how to improve the effectiveness of your development teams? How to be more productive? We have heard people wondering about it a lot and there seems to be millions of books about it.

One solution we see is having full-stack development teams. The concept is similar to agile cross-functional teams and stream-aligned teams, which are mentioned in Mathew Skelton and Manual Pais’ book: “Team Topologies”. The expression came from ‘full-stack developer’, but it does not mean the same thing. A full-stack developer is someone good at both backend and frontend development. However, the full-stack development team has all the knowledge they need to build this specific product (or part of the product or *business stream* etc.). This includes backend, frontend, mobile development, DevOps, UX, data analysis, business understanding, and so on. That doesn't mean everyone is an expert in all of them, but a **shared goal and vision** are essential parts of the concept. It means that everyone is aware of what is going on in different areas and what challenges there are. Like the ball that never drops and always stays somewhere in between, each team member needs to have a feeling of ownership. **You do not need to wait behind other teams so there is major growth in effectiveness and productivity.**



Of course, it means that the size of the team matters. Just like Scrum and several other sources state, we also believe that 3 - 6 dedicated full-time employees is the perfect size for a self-organizing team. If it grows bigger, being aware of everything that is going on becomes too time-consuming.

Another important aspect is that **there should be no knowledge or skill that only one person can master.** Then the stress and risks are too high. How can one go on vacation? How does one not become a blocker inside of the team? Sharing goals and knowledge **grows productivity** and makes **prioritization** much more effortless. Once the priority changes, the whole team is aware of it and there is no need to coordinate it between several teams (as would be the case if backend and frontend developing occurred separately).

A case study from our own experience - we used to have an infrastructure team taking care of the production environment. Now we have a platform team that builds the environment where all of the development teams can handle production activities with Kubernetes themselves.

In 1:1s, we have received great feedback from developers that this has been a remarkable change because:

it gives strong ownership feeling
it makes the process quicker
it is fun to learn new things

Of course, you cannot always have all of the knowledge in the team. If necessary, the team can ask for support from the outside, but getting it and implementing it stays within the team's responsibilities. That's ownership. And that makes everything else described in this book possible.

KPIs to measure:

the team's size and time spent in a month waiting on blocked issues

CHAPTER TWO

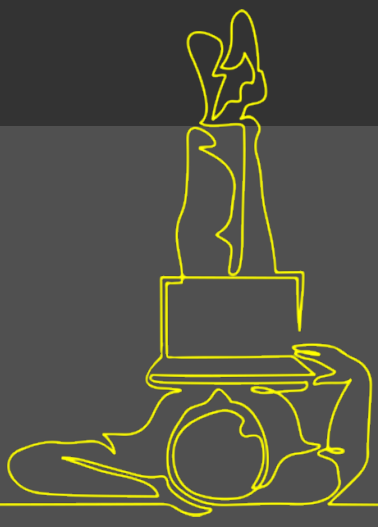
BUSINESS GROWTH

As discussed in the previous chapter, full-stack team members know the business value their product brings like the back of their hand. Each person in the team knows the answer to the ‘why’ question of every functionality they build. And we do not mean ‘why’ the product owner wants it, but more so why the product actually needs it. Is it a must-have, highly valuable feature that helps the product to grow and bring more business success? Has it been validated before putting a massive amount of effort into its development? How do you measure the success of it afterwards? And finally how do you learn from it so that you can make changes quickly?

Innovation is part of success, but innovation requires wise cooperation. The full-stack team chapter discussed how to build the team itself with a shared business vision and understanding. It is also important that development teams work closely together with the business side (marketing and sales) and customer support.

There is a wise saying: **“One of the most damaging effects of departmental silos is that they slow down innovation that drives growth.”** We have seen and experienced it in our own case studies too.

Case study:



Example One

Challenging deadlines were declared outside of the team's hands, so the team needed to prioritize every small thing wisely. Unfortunately, they were not aware of the exact business goals for that deadline and they failed to prioritize everything as well as they could have. Later on after learning the details of what the sales team actually needed on that particular date, they discovered how it could have been done much better. We will discuss direct communication in more detail in the next chapters to come. Still, this example is valid for how important it is for the whole team to understand the business goal while developing. They can take ownership of delivering what is really needed and when it's needed. It is unrealistic (and ineffective) to expect that the business side and product owner can prioritize that at such a detailed level. Rather, their input is to explain business priorities.

Example Two

We were building a minimum viable product (MVP) for a small startup. We also discussed business priorities and adding growth into the product from the beginning. Still, at some point, traditional thinking kicked in – what if the startup was working together with the marketing team on some ads and forgot to share those with the full-stack development team? Luckily, we realized it early enough to say STOP and organized a meeting to discuss marketing and development together. Two things happened as an outcome. Firstly, the marketing team received awesome additional ideas for their ad campaigns since developers are very creative people too. And second, since they know the product very well, it is easy for them to brainstorm how to market it. Developers understood what functionality was needed in the product for that campaign and how to build it better.

Growth hacking

Those examples sound basic and straight forward, but still, you can see such things happening over and over again in startups and bigger companies. **A growth hacking mindset is needed for companies of all sizes** because either you are a beginning business trying to reach your first engaged users or you already have an existing product that is used by millions each day. Either way, you still do not want to waste time and money on the wrong priorities or wrong functionality. And you definitely want to learn from your end-users as early as possible.

> *Growth hacking blends product development, analytics, and online marketing to rapidly generate, prioritize, and test ideas*

definition taken from the book: “Hacking Growth” by Sean Ellis and Morgan Brown

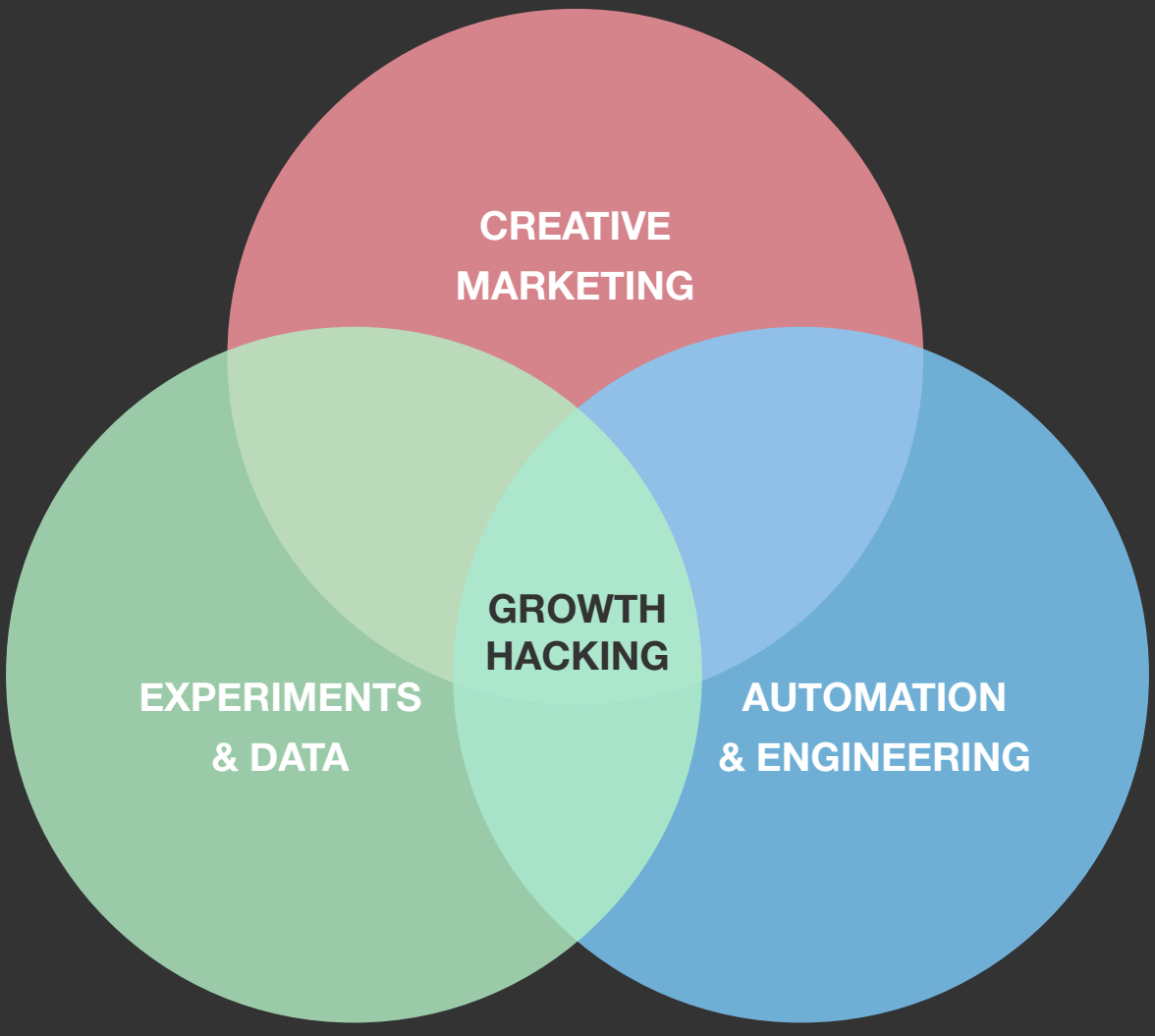
Taking action

Let's get into more specifics. What are the first steps you need to take in your full-stack team to bring more business growth?

1. Explain and assign business KPIs to the full-stack team. They are responsible for delivering the same KPIs as the rest of the company. Forget technical KPIs that pull the team in different directions. If there are several, then always remember to say which one is the Northern Star at that moment.
2. Always explain and understand the business' 'why' of every single feature and also the product's long-term vision.
3. From time to time, make the whole team walk in the shoes of the end-user. Depending on the product, it could be essentially stepping into the user's shoes or personally meeting the real end-users and interviewing them. Yes, it is mostly the product owner's job, but every developer in the team should be included and present once in a while.
4. Plan how to measure the progress and success of each feature. In addition to monitoring the technical statistics, the full-stack team should follow how the product is used daily. Of course, sensibly - measuring the things that truly matter. **A metric that means nothing for one company may be another's core growth lever, so choose wisely.**
5. Trust them to make the right decisions in prioritizing once the development team has done the first 4 steps. Allow them to brainstorm new features that bring more growth and support to sales and marketing.

How many of the built functionalities have a measurable hypothesis set before it is developed?

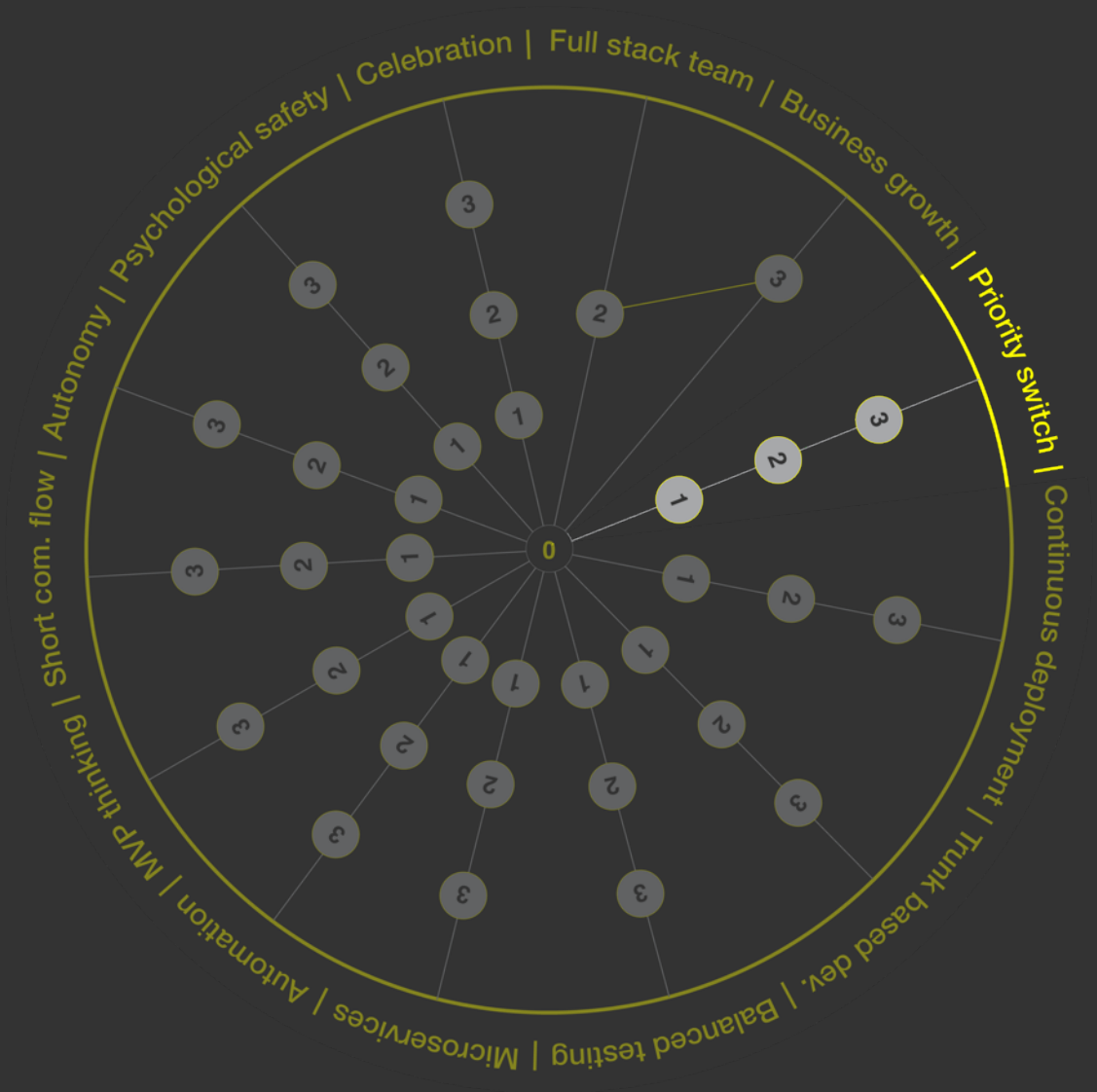
How many of the features that turned out to be not-so-successful could have been prevented with better cooperation and idea validation beforehand?



The more people are committed to business goals, the easier it is to reach them.

CHAPTER THREE

PRIORITY SWITCH



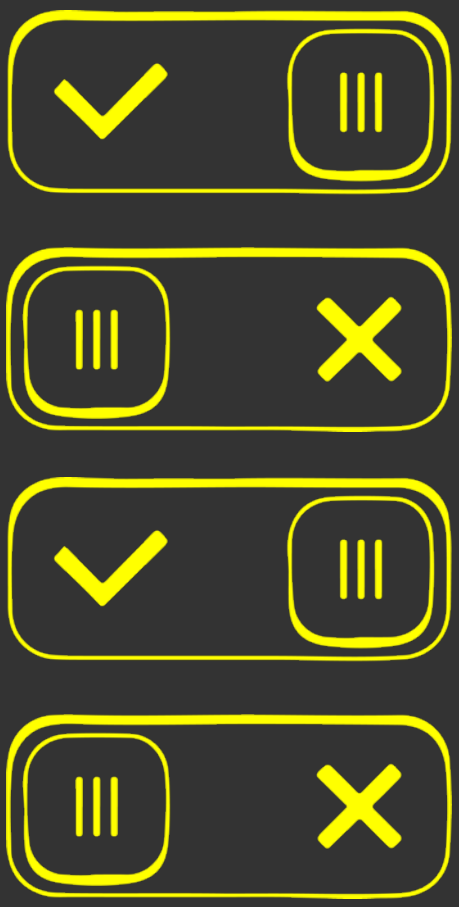
We live in a world where things are always changing. We learn from yesterday's actions, new opportunities appear, or COVID-19 starts to spread worldwide... For startups and growth companies, it is essential to use these opportunities as they come. This should happen very smoothly in development teams as well or you may lose valuable time. If a change of direction is needed, then it should be done now and not in the next sprint that starts in two weeks.

Processes and sensible mindsets can help you react quickly and wisely in those situations. If the team is aligned by business KPIs (and the Northern Star) as explained in the business growth chapter, it is easy to change their work priorities once they are notified of any changes.

Understanding why there are changes and having them part of a daily routine, makes priority switching less stressful. **It also saves time and increases commitment**.

Naturally, technical architecture needs to support making switching easier as well. The next chapters will give a detailed overview on how to do that with a minimal time-to-market mindset (unified with continuous deployment and trunk based development).

One action the team can do is set up their working process to be ready for constant changes. It could be through discussing changes during a standup or assigning one person each week who is responsible for quickly reacting to changes. It doesn't matter what it is exactly as long as it is agreed on beforehand.



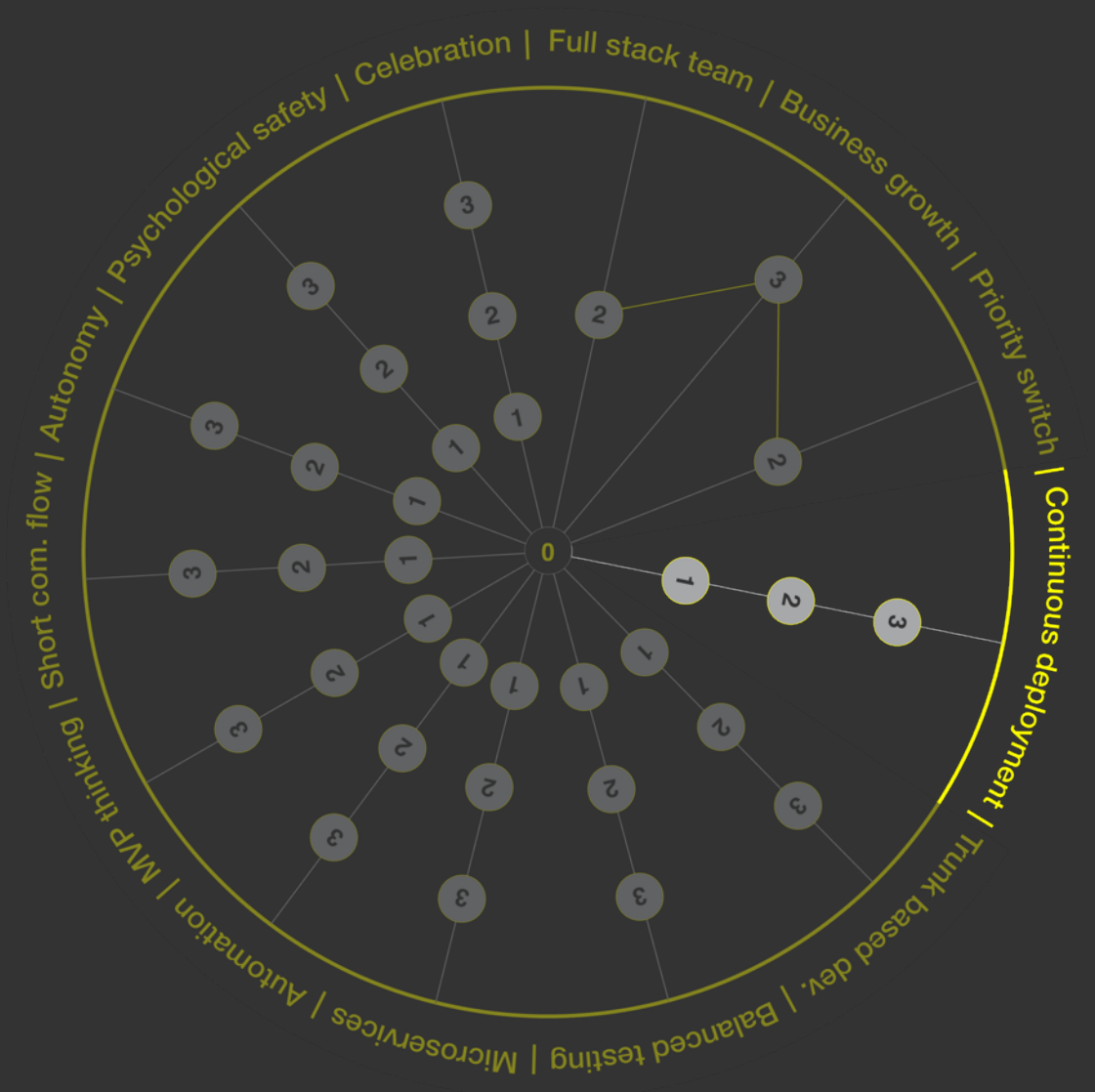
In the past, we've had a situation where the whole team concentrated on releasing a new product in one specific European market. Instead, a complete priority switch to release it first in the Arabic market came up, which meant an entirely different culture and language. It was a crucial business opportunity and the team was able to go through the day switching all priorities stress-free.

How much time is needed for switching priorities?

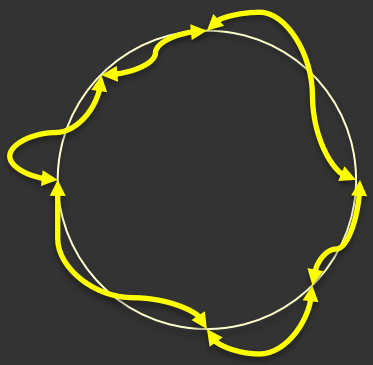
How stressful is it for the team members?

CHAPTER FOUR

CONTINUOUS DEPLOYMENT



Continuous deployment (CDP) is a software release process where changes are automatically deployed to the production environment without human interaction. It relies solely on automated testing and an automated release pipeline.



Releasing as often as possible encourages you to validate the smaller changes to your users which allows you to adapt quickly. With more experiments like these, your innovation and business will **grow exponentially** while learning quicker than competitors.

Every line of code written by a developer only starts to bring value once it finally reaches the end-users.

The delay of manual releasing itself is keeping a business from achieving the goals that the line of code was meant to convey.

The smaller the increments of change, the less risk of breaking things. Which in turn, means faster recovery from failure.

Continuous deployment gives **better software quality** by forcing best practices in testing, monitoring, communication, and developer mindset. When there is no manual testing, the developers protect themselves from mistakes by writing tests that matter. They write business logic that they understand because there is no one manually verifying it before it reaches the end-users. This forces better communication between business and development. When changes are in production, there is no false hope that everything is always OK (it never is - even if you test manually by 10 testers) - so you have to have a good monitoring setup to detect problems early, possibly even before the users can report them. Continuous deployment is the trigger to business growth.

Continuous deployment means to rewrite your processes.

Start by implementing CDP on a technical level. Set up pipelines in your CI tool, making all master commits to run automated tests and deployments. Add basic monitoring to catch exceptions and performance problems.

Now let everyone know how it works and encourage them to try it out.

Start to execute this approach by introducing it to both business and development. It is imperative to involve development as early as possible so they can automatically start writing the necessary tests. Then, you can continuously improve the monitoring and testing. And lastly, check what the end-users are doing.

How many deployments do you do each day (per developer)?

How long does it take from code push to reach production?

Case study:

We were developing an integration to a third-party system. We had the basics all worked out and were having a meeting to discuss the last details. Once we agreed on the details, we started to talk about when we would implement them. As it was a very high priority mission, we said “Ok, we’ll do the changes right now with a few hours of coding. So, you can expect to see the changes in production in a couple of hours.” The third-party was having a WOW effect, “That’s so fast!” they said. They started to talk about sprint release and how previously they would only be able to release it by the following week.... a week lost for their business. And that’s only this time. If they need to do additional changes, it will be another week. And the same for all other integrations they were doing. It’s a waste of time.

Another example is when we had a team many years ago that was struggling with releasing quality products. In theory, everything was fine: sprint, quality assurance, release management, etc. However, there were also a lot of problems. DevOps reports started to show that the team should release more often to increase the quality. And that is what we did. Overnight, we went from releasing once per month to releasing multiple times per day. And it worked: the quality **increased**.

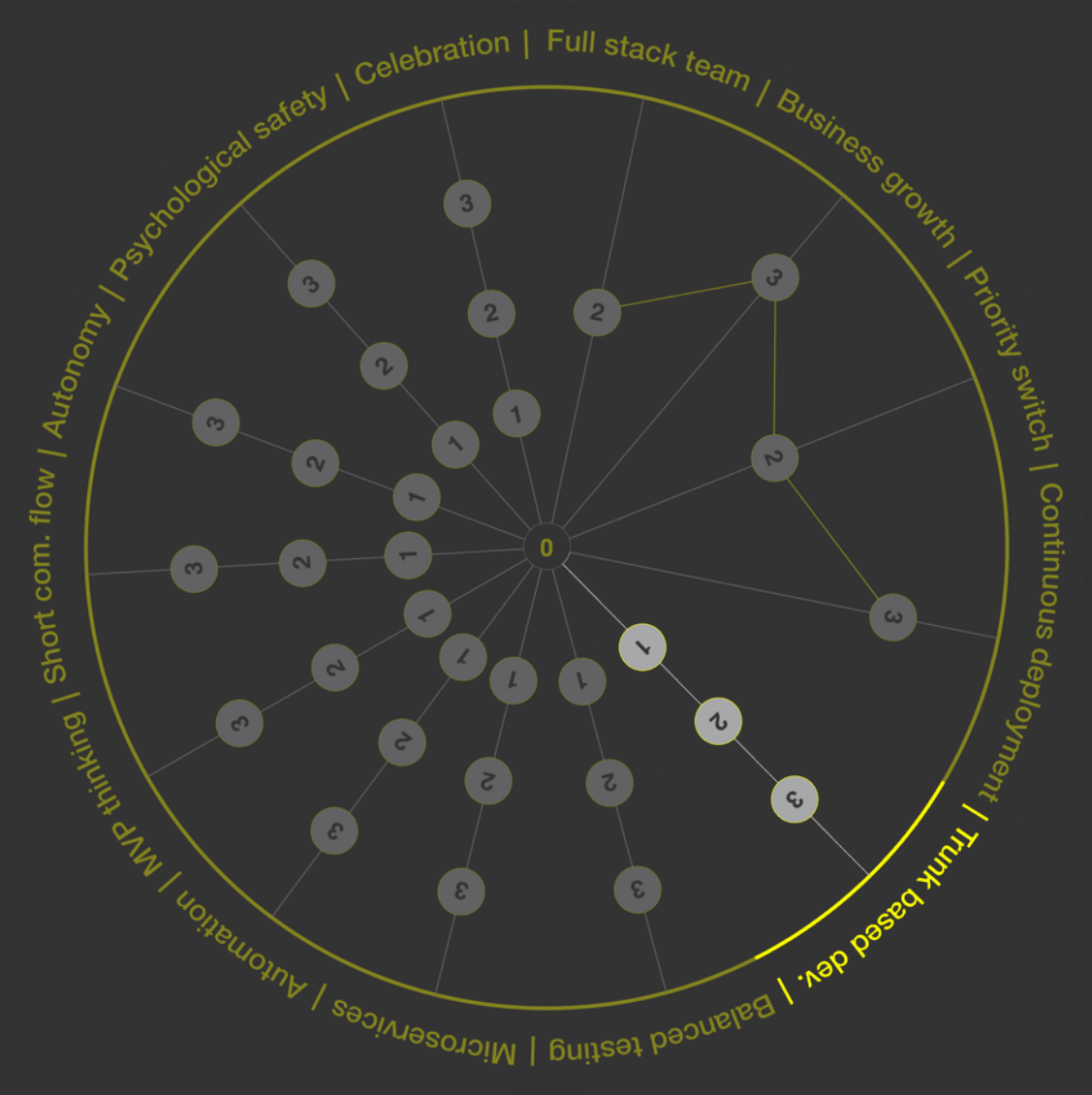
There were two key elements that resulted in this success:

Developer mindset. If you write code that goes to production just after you push it, you will take another look at your code, test it locally, and you won’t just throw it to the QA. It was a game-changer.

Quality of automated tests. It is considered essential to have a good test suite to start practicing continuous delivery. In this case, it was the opposite. Once CDP came, the tests magically became relevant. We knew there was no manual verification anymore, so developers had to write tests to protect themselves.

CHAPTER FIVE

TRUNK BASED DEVELOPMENT



A source-control branching model, where developers collaborate on code in a single branch called 'trunk', resist any pressure to create other long-lived development branches by employing documented techniques. They, therefore, avoid merge hell, do not break the build, and live happily ever after.*

** master, in Git nomenclature*

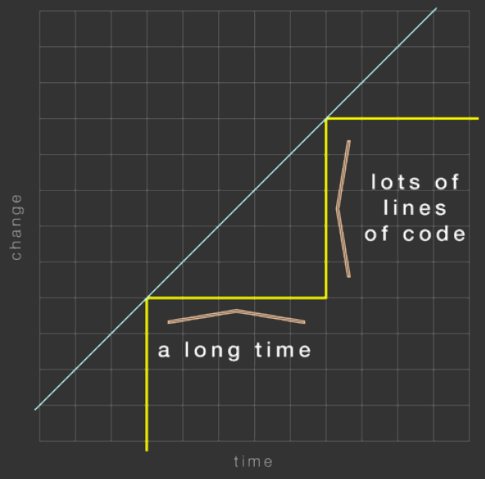
Source: https://trunkbaseddevelopment.com/

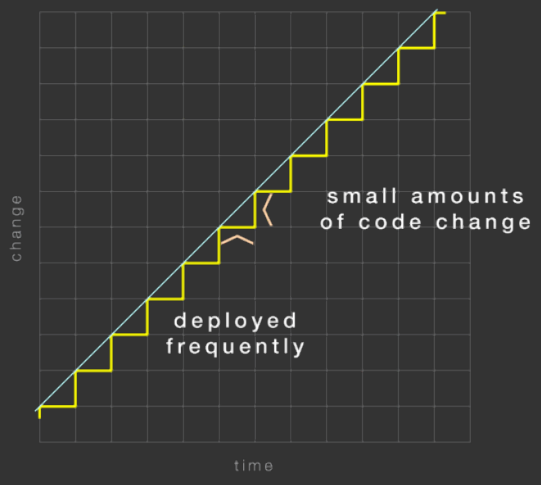
We like to keep the **feedback loop from end-users to developers as short as possible**. For that, we need to practice continuous deployment (CDP). Trunk based development is the fundamental cornerstone to pull off these frequent releases.

How it allows us to release faster

When doing smaller commits often and releasing them as soon as they are ready, we minimize the change failure rate and mean time to recovery (MTTR). It has been measured that in total you get **five times fewer problems when you release more often** vs only once per week. And if you get into trouble, then MTTR is up to 100 times faster*. If you think about how this is possible, the key here is **small commits**. If you release one commit at a time and they are small, then it's more likely that the developer did it correctly rather than writing code for a week and hoping that all of it works in the end. Also, if you release a hundred commits at once and something goes wrong, you have to start investigating which one of them is the faulty one. It takes time and your MTTR is suffering. If you release one commit at a time, it's obvious where the problem is and you can either revert or fix it quickly with the confidence that nothing else is affected.

*State of DevOps Report 2017, Puppet Labs





Drawback of merging

Now let's talk about merging. There are two key points to address and the first is merge hell. If you need to merge two branches that have significant differences, it's a pain. With smaller changes, this problem becomes almost non-existent, which increases the developer happiness factor. However, when you have multiple long-lived branches, it might mean writing code on wrong assumptions. Someone else might have already changed the code on something you depend on and it creates misunderstandings that result in bugs. It's also common that if you release on a fix branch and commit straight to fix branches, you sometimes forget to merge it back to the mainline. Again, then you're in trouble. **Working with one long-lived branch avoids both of these drawbacks.**

It is also common that the branching model is a copy of what is already represented in your organization structure and bureaucracy model. If you are in such a situation, you can start by modernizing the organization from down to up. And reason through the benefits of doing smaller changes at once. It's even possible to change corporates to become more agile.

GitFlow is a horrible match for an agile company and has unfortunately driven change away from the developer's perspective. Therefore, it creates a less efficient organization.

Don't just do it because it has been done like this for years, question the *status quo*.

Master vs short-lived feature branches

It's possible to practice trunk based development with either committing straight to master or doing short-lived feature branches where you create a pull request to master when a small batch of code is ready. First of all, the choice between these two is less important than the choice between trunk based development and GitFlow. It's crucial to consider which one of the trunk based development forms is the right fit for your business.

The general rule is that if the team is small and the codebase is understandable, then it's OK to commit straight to master. However, these days teams rarely grow beyond 5 to 6 members anyway. And with microservices, the codebase is more isolated so we have found that size is not the best way to describe the difference here.

The vital difference between them is how DevOps-y you want to be. If you push code straight to master, you can follow the CI/CD pipeline and check the database/logs as part of doing that particular task. This is the shortest feedback loop possible. You can check right away to see how your code is working on the end-users. It's like a sports car - you get to be in direct contact with the road.

Techniques to practice

It's not always possible to create valuable pieces of code for your users within a matter of hours. It is, however, still beneficial to keep the commits small as explained previously.

So, what should you do?

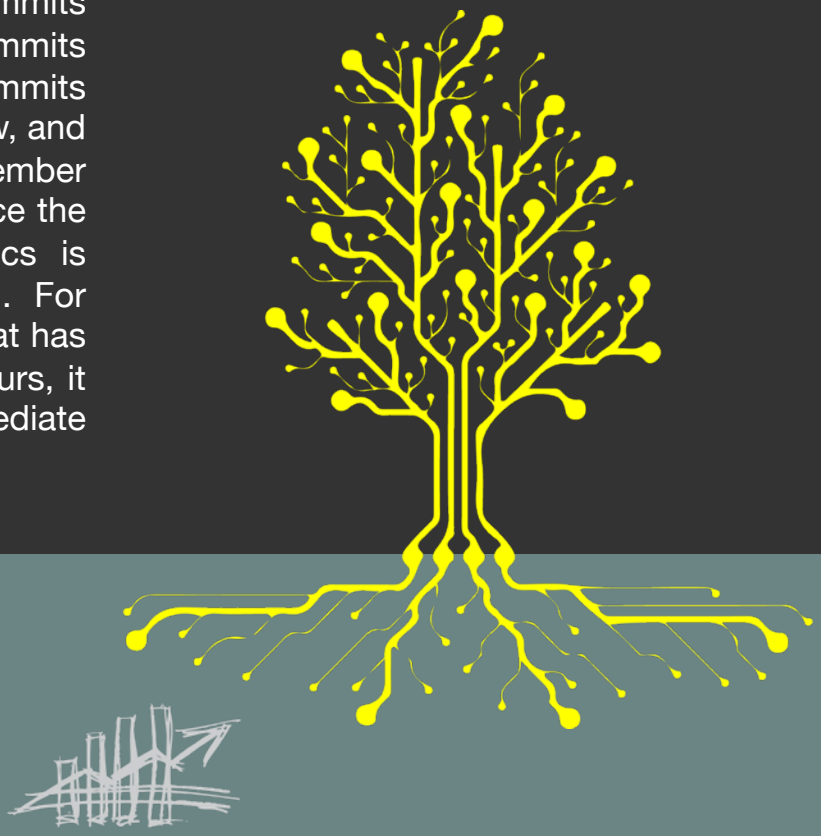
Feature flags. An "if staging" or "if false" code block is better than having a separate branch. Everyone sees this code and can integrate into it.

Branch by abstraction. When doing significant changes, still do them in smaller batches, but create an abstraction so everyone can see that there is another implementation coming.

Pushing straight to master also challenges a bit of how to both force code reviews for all code made and how to actually review the code. With pull requests, it's easy. The code won't land in master without review. However, with straight to master, we need tooling for that.

At Concise, we use code review dashboards that track all commits. The dashboard will show the commits that are without review, the commits that you have to review, the commits that you have to fix as per review, and also how many reviews each member is doing (so you can load balance the reviewer). If one of the metrics is turning worse, it will turn red. For example, if there is a commit that has not been reviewed within 24 hours, it turns red and will then get immediate attention from the developers.

Another challenge is grouping the commits for a review so you don't have to review multiple, very small commits. Instead, it will be only one issue at once. For this, we are using Upsource, a handy tool by JetBrains. This tool is also what gives us dashboards. There are other options as well, but these are mostly dedicated code review tools (and you can't just rely on Github).



How long does it take from code push to production?

**If you have short-lived branches,
what is their mean time of living and how do you
detect if they are short-lived in practice?**

Exceptions

Your team gets a new developer - should they still commit straight to master? Well, it depends. It's quite beneficial to do pair programming with new devs (at first) so they get to know the environment and the true spirit of the DevOps you have. However, sometimes it's fine to review the code from a short-lived feature branch as well. It's up to you to decide what is reasonable.

Say you are developing a mobile application or some kind of device where you don't control when the user is about to update the code. Or worse, if there are more restrictions in between such as the slow review process from the Apple App Store. Here you can still follow the general rules of trunk based development, but you might want to create release tags every time you send the new version to flight.

Developers

When a new developer joins Concise, we are accustomed to the fact that they may be used to doing branches and they want to continue doing so. We stay calm and take it easy. It's okay that they might be a bit afraid of the new way of working. They will still get a warm welcome and time to make themselves comfortable. For some, it takes a month to see the benefits of our approach. But for other people, sometimes it takes half a year. However, once you get used to it, you won't want to go back to branches (especially GitFlow). No one does. We have multiple testimonials from our developers who, in the beginning, could not believe the way we work. But once they switched over, they actually became strong advocates of trunk based development.

CHAPTER SIX

BALANCED TESTING

We aim for optimal time spent on testing.

Why is it important?

Testing 100% of every single thing is impossible. Testing close to everything is expensive. Almost no business can afford it. The important thing is to find a balance between what to test, how much to test, and which way to test. This is so that you spend the minimum time possible on it while still guaranteeing maximum quality.

How should it be done?

Do it through a combination of testing pyramids and monitoring.

Tests

For backend, we usually start with integration tests. On a technical level, it means that we spin up the application (think SpringBootTest) and mock all external integrations (think Wiremock), but use an actual database in Docker. Then send JSON to the API and wait for something back. This covers the API for backwards compatibility and as a benefit, gets the code coverage relatively high as well. This tests most of the happy flows. Next, we add unit tests for each bit of complicated logic that was not tested by the happy flow. **The balance between unit and integration tests depends on the stack used** - some systems are easier to unit test, but if integration tests are convenient, it's beneficial to start from there. To make sure that the happy flows work in the actual environment, we create a few of them in a staging environment. This is similar to those used in the integration phase and gives us the confidence to use CDP.

For frontend, we unit test the logic in React components. In the beginning of the project, we also test manually many times. As the project matures, we add some automated end-to-end tests using tools such as Selenium. These tests are to cover the absolute minimum happy flow that lets us know if it is possible to log in and proceed with the most important flows.

For apps, it's similar to frontend testing. Unit tests are rudimentary for logic and end-to-end tests as well. Since it's difficult to release apps as fast as we would like to, we need to practice **gradual roll-out.** If possible, we prefer Progressive Web Apps because it's easy to update as we go.

Monitoring

Now that you have an idea of how the tests can be balanced, always consider that there will be problems you can't detect with tests. For this, we have **extensive monitoring** in place.

Four steps you should prepare

Test what is reasonable since the unthinkable will happen anyway - and when it does, you want to catch it before the user reports come in.

How to be prepared:

1. Apply a zero-exception policy that detects all application problems. Alert yourself immediately and find the root cause. Most logging stacks have a feature to send notifications based on the search string, such as “exception”. Sentry has created a service out of it – give it a try. Catching intermittent issues early saves you from bigger downtime later on. So, remember to continuously check each and every one of these issues.

2. Monitor performance in different layers of your system. Put emphasis on the bottlenecks; database monitoring can be difficult, but it is essential.

3. Add end-to-end tracing. Check the anomalies.

4. Use synthetic API / interface monitoring to detect happy flow problems. Those test if users can log in, view, and click the main buttons. If this is failing, you know there is a critical issue with the system.

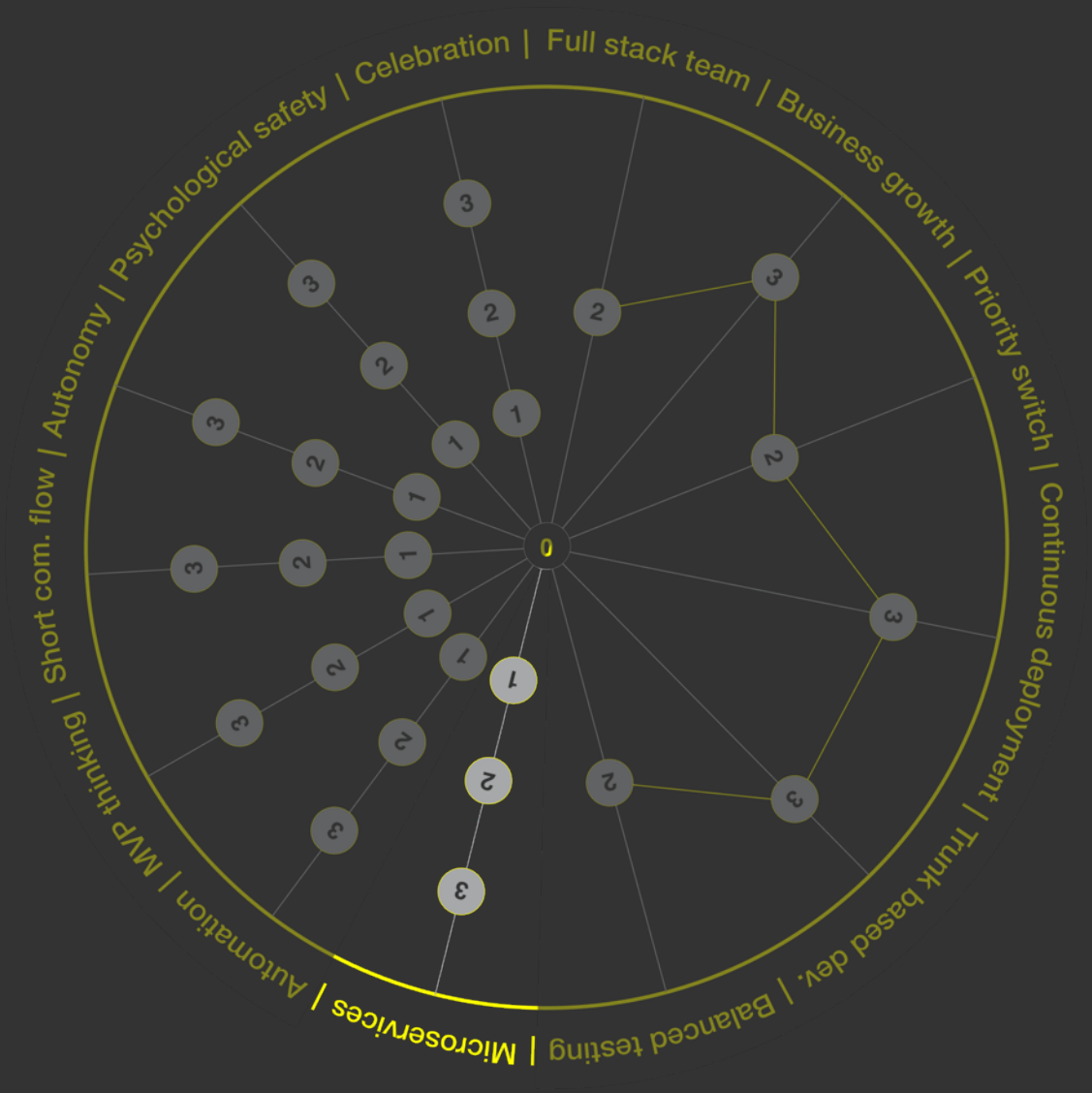
All of these will trigger alerts to developers - it may be a Slack notification or an automatic phone call, but it will allow developers to respond quickly.

An important part of the testing strategy is to understand when it’s beneficial to test and when the monitoring will be more effective. However, you will need both anyway and both of them are critical to use together with CDP.

So in the end, testing and monitoring will give you the best quality and speed for the money.

CHAPTER SEVEN

MICROSERVICES



As organizations grow, so does the software they produce. The classical way to evolve software is to keep adding new features to the existing one. This is called monolithic architecture.

Microservices, on the other hand, are **small, autonomous, independently deployable and releasable pieces of software that work together via well-defined APIs**. This means that instead of a big monolithic application, we have many smaller components. At first glance, it might seem more complex to have more moving parts. Microservices indeed introduce their own set of challenges, but they help you grow your organization and provide value to your customers faster.

Organization alignment

One of the problems we see with monolithic architectures is that they **hinder growth**. As we scale our organizations by introducing new teams, we can see that people start to step on each other's toes. **More coordination** between different parties is needed when working on new features or performing releases. Even though the teams might be independent on the organization chart, they all have a single integration point - the monolith. Essentially, the software architecture doesn't reflect the organization's communication paths.

Microservices enable you to align your organization's architecture with your software architecture. Instead of having a single integration point where all teams must meet, every team can have their well-defined areas of responsibility.

We've experienced this with our customers where we have several teams, each containing 6 or 7 members. They work together, but have their own microservices that they're responsible for. Teams aren't coupled to a monolith anymore. They're more independent, autonomous, and can release software without having to coordinate with other teams. This means we can ship features faster and provide value to customers sooner.

Resilience

A bulkhead is a wall within the hull of a ship that also creates watertight compartments. In the case of a hull breach, bulkheads stop the water from moving from one section of the vessel to another, containing the damage. The same analogy can be used when comparing microservices vs monoliths. **Issues in one part of the monolith can take down an entire application**. On the other hand, microservice boundaries can be designed as bulkheads. **We can contain a failure in a single service** and it won't affect other parts of the system. The system as a whole might experience degraded functionality, but parts that are not affected can still provide value to customers.

Microservices enable us to design an architecture for low-risk releases. There's always a risk when we make changes to the system. But if we can contain changes to a small region in our architecture, the **potential blast radius is smaller**.

Fast Feedback

An essential part of writing software is how fast you get feedback. The faster we can detect issues, the easier it is to fix them. Microservices are small compared to monoliths. This means that the time it takes to build, test, and deploy them is also shorter. **We can get our code deployed faster and issues in production systems can be fixed sooner.**

For example, it could take an hour to build, test, and deploy a large monolith. Imagine you need to make a small change to the system in order to fix a critical production issue. Your time to recovery is at least one hour. That's assuming you detected the problem immediately when it occurred and had a solution for it ready. But in reality, time to recovery would be longer than one hour because it takes time to react, investigate, and develop a solution. Oppositely, microservices can be built and released in minutes.

Even worse, large deployment times lead to **infrequent releases.** Since we feel that going through the long process of releasing the monolith multiple times per day is painful, we tend to collect changes into a single batch and release them together. This has a counterintuitive effect of making every release that much riskier.

Conclusion

Microservices aren't a silver bullet and still come with their own set of challenges. However, we believe that the benefits outweigh the costs for setting up and maintaining many independent services.

CHAPTER EIGHT

AUTOMATION

Automating manual work saves time and raises developer and customer satisfaction. By automating tedious and repetitive tasks, we can spend more time on important work that's **meaningful** for our business.

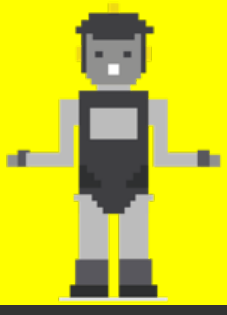
Manual, repetitive work is slow and error-prone. Often, this work gets delegated to dedicated departments.

For example, software testing is the QA department's responsibility and application deployment is assigned to operations teams. By doing that, we unintentionally create dependencies between teams. For instance, engineering teams have to wait until QA tests their work or new features aren't shipped until operations are ready to deploy them.

With automation, we can move the work closer to the source. We can give product teams more autonomy and help them get feedback faster. Automation enables us to ship features more quickly with less cost and improved quality.

Automation is not a checklist of things you have to accomplish. For us, **it's a mindset**. It's about noticing manual, repetitive, and automatable work that comes up over and over again.

> *If a human operator needs to touch your system during normal operations, you have a bug.*
>
> *Carla Geisser, Google SRE*



CHAPTER NINE

MVP THINKING

MVP = Minimum Viable Product

Since it is a common term in the startup world, we will not explain it in detail. If it's new for you, we suggest reading "The Lean Startup" by Eric Ries.

His **BUILD > MEASURE > LEARN** cycle is an approach you have seen in this book for several chapters now. We talked about measuring in the business growth chapter and getting rapid feedback in the more technical chapters like microservices, CDP, and trunk based development. So, when we talk about MVP thinking in #superagile, we don't mean building the product's first MVP. We talk about the mindset of **always thinking about the smallest part that starts to bring value** and is measurable. It doesn't matter if you are just starting your journey or you have millions of users already. There is also a term called Minimum Viable Feature (or MVF), which would be more correct in this situation, but let's keep it simple. We are talking about **the way of thinking**.

But why?

Why do we want to build a technical architecture that enables quick feedback? Why do we want to spend time figuring out the smallest part? And why do we want to measure everything? Because that is the secret highway to business growth as mentioned numerous times before.

It's the best way to learn about your users and truly the best way to avoid spending time on things that are not needed. It all starts with thinking. **"Dream big, start small"** is a motto we use. Once you master MVP thinking, everything else described in this book easily follows.

Learn as early as possible

If you plan to add a new feature, maybe the first thing you can add is a button in the product with no actual functionality. When the user presses the button, they get a message saying this feature is coming soon. At the same time, you learn how many users have pressed this button. If none, then why spend time building it? Either the button is in the wrong place or there is no interest in that feature. You merely added a button, measured how users reacted to it, and learned from it.

Challenges

We tend to assume that everyone thinks the same way we do and that we know everything. Or at least the product owner does. **The reality is that we don't know everything**. Humans are unpredictable.

In practice, MVP thinking can be challenging. The example with the button seems simple, but often the situation is more complex. Does beautiful design matter? Does adding edge cases matter?

Is it possible to break it into smaller pieces? There are no straightforward answers; it all depends on the users and the situation. But the more you practice, the better you get at it. Since we love our product, we want to offer the best and always add a little extra...

But that's a habit worth getting rid of.

Here is a case study of a discussion we have heard so many times when building mobile apps: should it be available for both iOS and Android from the very start?

Our first reaction is always yes. But how about doing some research on which is more popular with potential customers and then first concentrating on that? Well, you can already start to learn and gain feedback from iOS users while bug fixing the Android version.

Again, sometimes it's more complicated. The app or feature might be for some team activity and there's a high probability that people in that team have different phones. So then the value of the product or feature comes from the team using it together.

So…it depends.

Checklist:

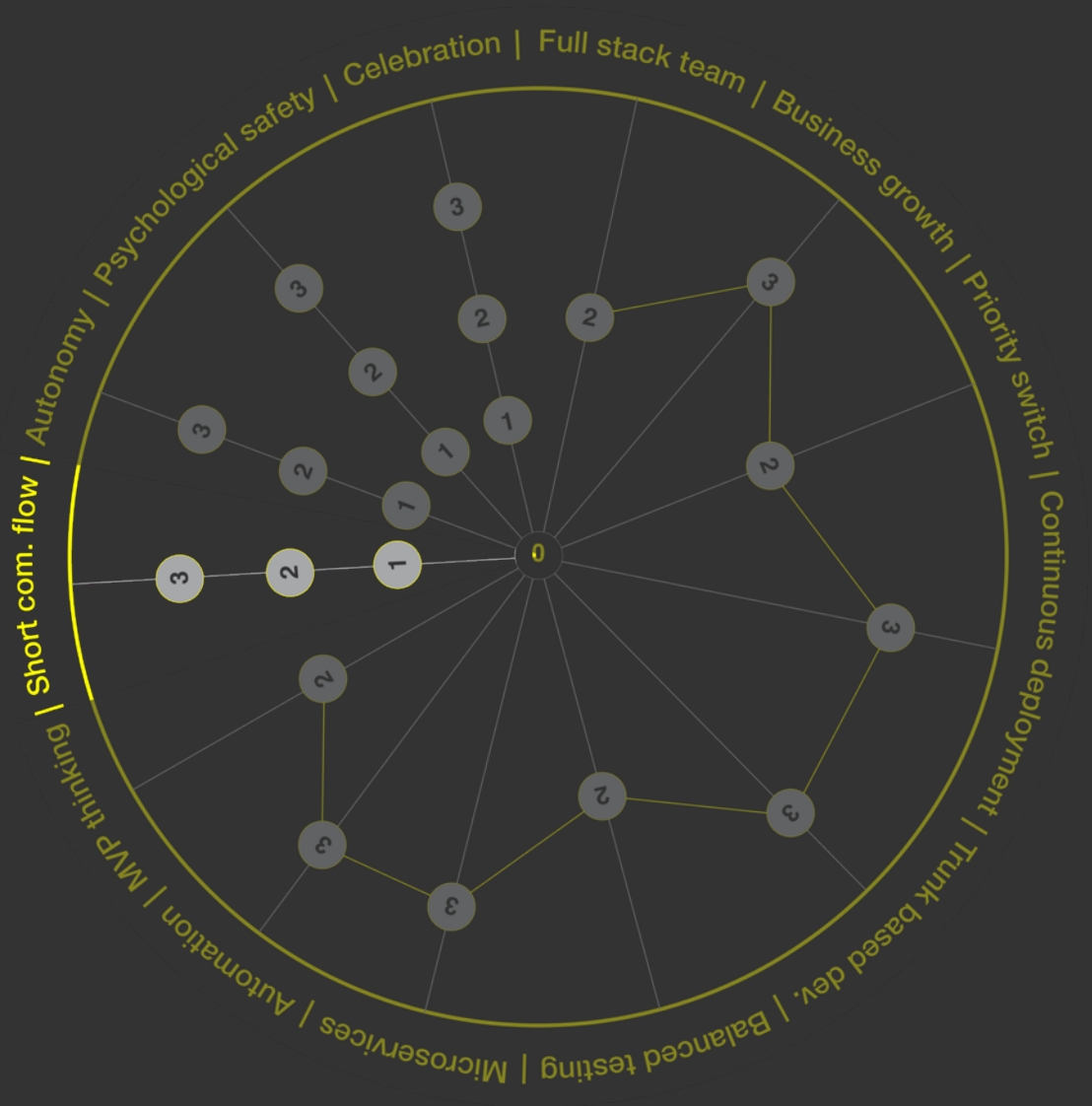
- ✓ Before you start any development, have you done all that is possible to make sure this product / feature is needed? Do you understand the pain that goes with it? Have you done the proper questionnaires or even interviews? Paper prototype testing? **Question everything you feel sure about.**
- ✓ Make sure you plan out how to measure. Otherwise, you might end up not having the data to learn from.
- ✓ Use MVP thinking in every step of your development process.
- ✓ Take time to learn. As you remember, switching the priority is welcomed in #superagile teams.

Measure how quickly you go through the whole cycle – from starting at the initial idea to learning from data on average.

In most cases, it should be less than a week (unless we talk about building a completely new product from scratch).

CHAPTER TEN

SHORT COMMUNICATION FLOW



Communication faults are expensive. Yet, they happen often. Task managers who translate business needs into detailed tech tasks tend to end up in situations where real needs get "lost in translation". You need teamwork and not individual experts who do their thing and don't care about the rest. We shared an example about the team who did not understand the sales team's needs in the business growth chapter. **That is why reducing the risk of communication faults can save a lot more money than a 10% cheaper hourly price.**

In our #superagile approach, we are always looking for a better balance between different roles and direct communication. Of course, it would not make sense to include everyone everywhere - that is a waste. There are different roles such as people lead, product lead, tech lead, product owner, and so on. However, those **lead roles support the team to achieve their goals, not to play telephone in between.** If one person becomes a blocker, you need to reorganize your communication.

Quick test-question: "do all developers in the team speak directly to the business side?" Our case study shows that this change can boost up effectiveness even in big corporations. Business side people are much happier as well - their needs are met and they are better aware of the progress and challenges.

Direct communication is essential for using continuous deployment. Continuous deployment supports business growth. But one of the main excuses developers use to avoid it is whether or not they can be sure that they have built the right thing. In that case, why did they spend time doing expensive development if they were unsure? Direct communication is crucial here. Improve that if needed, but keep continuous deployment.

Always make sure communication channels and meetings are planned wisely so that it won't overburden anyone. Discuss how to improve it at each retrospective. Here are the three most used tips from our #superagile teams:

1. Use multiple Slack channels.

Create a new one for each noteworthy subject and involve everyone who should be aware of it. Avoid direct messaging.

2. One person in a team is responsible for answering business questions.

Switch that person daily or weekly.

3. Product vision and goals should be overcommunicated rather than under.

Keep it simple!

Less time waiting on others means less information that gets lost.

How often does someone in the team say:
" this could have been prevented if we knew that beforehand " ?

CHAPTER ELEVEN

AUTONOMY

Autonomy is tricky. It's impossible to say that one team has full autonomy if it just exists in a bigger ecosystem.

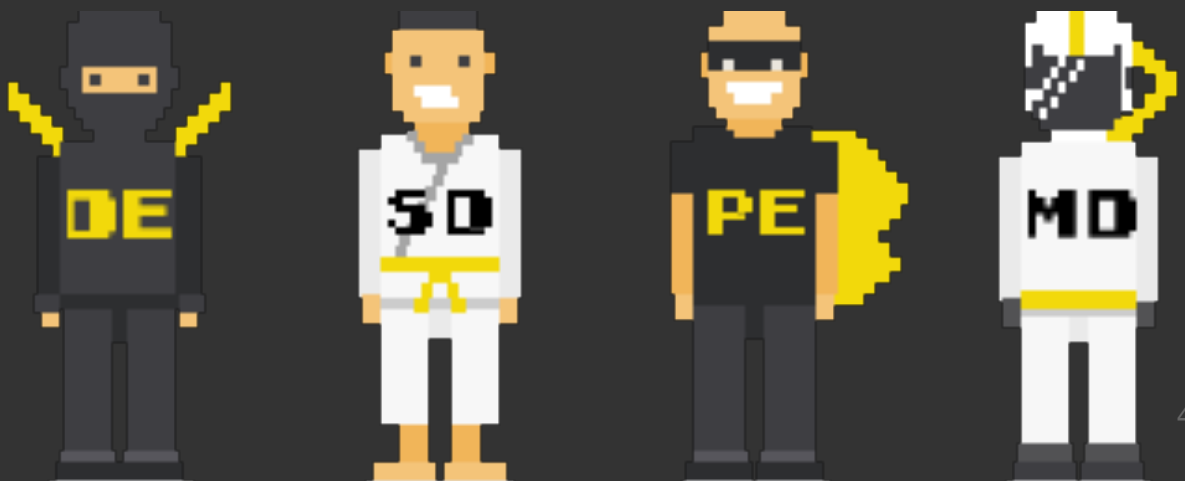
Nothing will work without good cooperation. As mentioned quite a few times, even closer collaboration between developers and business people enables more growth than in traditional organizations. All full-stack teams align according to the Northern Star and each team cannot choose different goals. **Moving in the same direction is necessary for growth, but so is understanding the 'why**'. It becomes the team's goal and vision.

However, there are multiple places where teams can and should be autonomous. You already read from the microservices chapter that good software architecture gives the team some autonomy. Why do we say that #superagile is a framework or culture and not a process? Because we do believe that the effective detailed process depends on each specific team. What works for one might not work for the other.

It's all about creativity, inspection, and adaptation.

We have created a 3-hour workshop about teamwork, which includes both the technical and process sides. This is everything starting from default workflow products up until the team practices retrospectives and culture canvas. **In agile, there's Shu Ha Ri, which means that first you follow the rules, then break the rules, and in the end, create your own rules.** Why so? Because **to understand the meaning behind the rules you should experiment with them**. Once you have understood the real reason, meaning, and learned that something does not work for you effectively, you can try to make changes.

Eventually, you are free to master your own rules as long as you keep the original need covered. For us, that means autonomy. Yes, there is a starting point, but each self-organizing team is unique and free to find their own ways to move towards the common goal!



CHAPTER TWELVE

PSYCHOLOGICAL SAFETY

We all know that keeping the team effective and productive is something difficult to create and easy to lose. Similar to building trust goes the atmospheric sense of psychological safety. Maybe you have heard of Google's study on effective teams. **Psychological safety** was the first and most important thing: we couldn't agree more.

If we feel safe to take risks and be vulnerable in front of our team, we can learn from our failures, overcome obstacles and challenges, and be creative in finding innovative solutions. Innovation, however, is needed for business growth (another element of #superagile). And so is constant learning.

In the technology world, new frameworks and tools are continuously released. You might lose valuable time and money if you always stick with using something because you have always used it.

Robert Kiyosaki has said: "In today's fast-changing world, it's not so much what you know anymore that counts, because often what you know is old. It is how fast you learn. That skill is priceless." But for learning, we need a safe environment.

So, the easy control-questions here could be: "do I feel comfortable enough that I can ask when there is something I don't understand?" or "can I speak up even when I am thinking differently from others?"

Psychological safety creates the culture of inclusion: a diverse range of ideas, questions, and solutions that give rise to the safety of being seen and heard. Though, it's a delicate feeling because it can easily be affected by the opinions of others or even questioning the decisions made by other roles in the company.

The trick is that safety starts with **being an example**. So, acknowledge your own safety first and then show and share it with others. Be present with your team. It includes the little things like mindful listening, noticing, giving support, or even asking questions to show you are there and care about them. It is done through presenting a non-judgmental approach to everyday challenges and praising failure. These aspects are part of an amazing journey toward finding the right things to focus on. But sometimes, even eye contact may be enough - instead of gazing at your phone or laptop during meetings, **look the other person in the eye to confirm that their message comes through**. During #superagile workshops, we leave laptops in another room or switched off.

In fact, our case study shows that after organizing company-wide failure pitching, the overall culture became more open and honest.

Notice if someone makes a mistake or the outcome isn't as expected
- do they share it freely or start to find excuses?

Notice in team retrospectives when a previous time period is discussed
- is time spent discussing who did what or on what to learn and improve?

CHAPTER THIRTEEN

CELEBRATION

Brené Brown has written:

> *We've got to stop every once in a while and celebrate one another and our team victories, no matter how small. Yes, there's more work to be done, and things could go sideways in an hour, but that will never take away from the fact that we need to celebrate an accomplishment right now.*

When we found ourselves in a situation where releasing was so stress-free and part of each day, we realized that we did not celebrate anymore. If you release once a month or once in half a year, it is such a big event that celebration comes more naturally. But we don't want it to be that way since releasing often brings more business success (as you have hopefully understood by now).

Besides, is it really just releasing *something* we want to celebrate? How about business growth? Or failures? Or great teamwork?

If you are creative, you will find millions of things to celebrate that are aligned with your business' mindset. Celebrating helps everyone in the team feel the value they have brought to this world. It unites and increases ownership.

Often people think that celebrating means going out together or having some kind of team event; however, it can be many other things too.

It can be a ‘thank you’ or praise at the end of a retrospective. It can be dedicated time to think of the achievements or laugh at the failures.

So once you have mastered all of the elements of #superagile, never forget this last one - **celebrate**!

Hi! We hope you enjoyed this book.

Good news!

Our app “Superagile” is out!

It will help you to conduct the workshop yourself for your team!

Get it from the App Store or Google Play.

Are you interested in redeeming a 2 hour #superagile workshop for your team that results in your individual success plan for improving productivity? Click the button below and let’s chat!

FOLLOW US ON SOCIALS

concise

* Book version 1.3

It will keep changing over time as we learn and experience more ourselves.