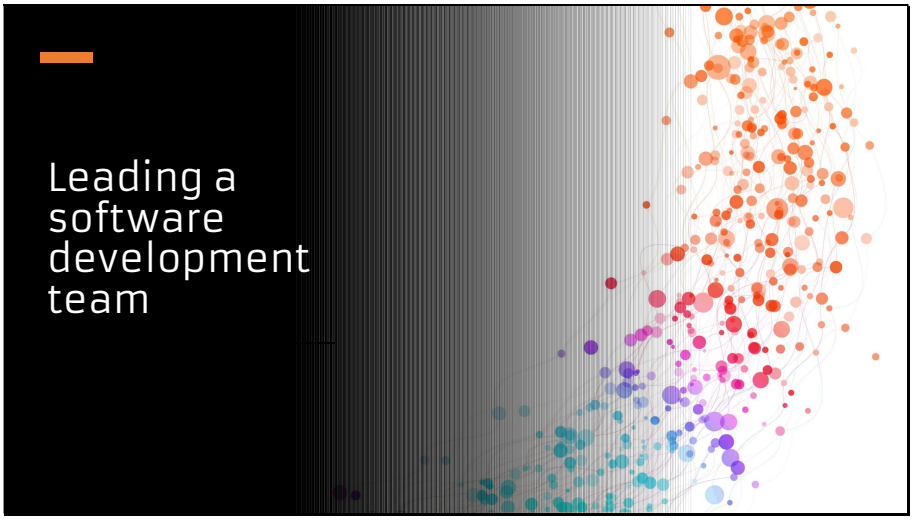
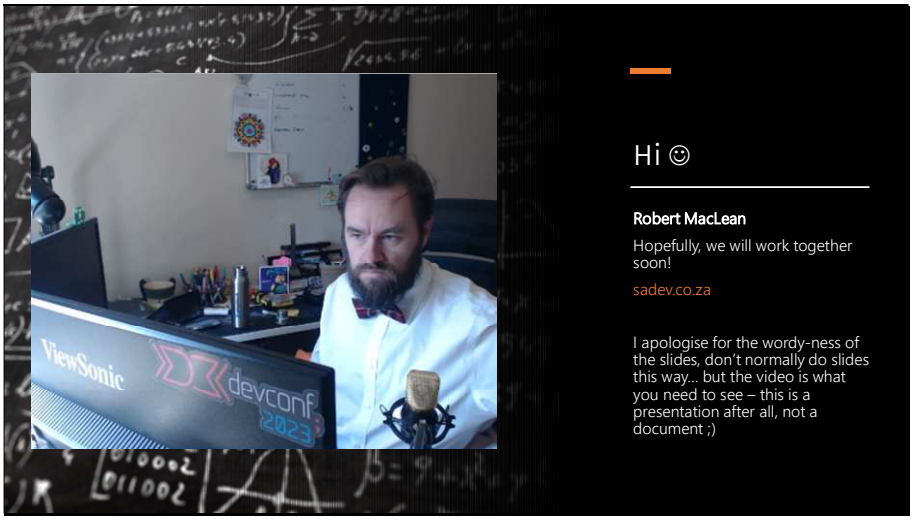
Slide 1

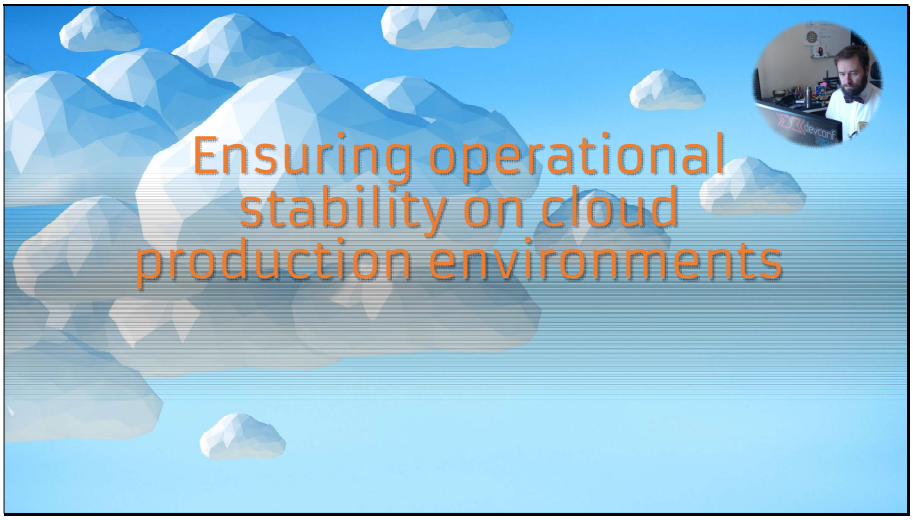


Slide 2



Hi, my name is Robert, and hopefully, we'll be working together one day in the near future. This is my presentation based on the slide deck I submitted. I'm not proficient at creating overly verbose slides. I don't come from a heavy academic background where slides are designed for post-class reading; rather, I believe in the value of the presenter, with the slides serving as supplementary material. That's how I've attempted to make them a bit more concise. I've also attempted to incorporate additional content into the notes section. However, I did think that for a presentation slide deck, I should conduct an actual presentation. So, hopefully, this video provides valuable material and offers insight into my approach. Let's now delve into the first section.

Slide 3



The first section for today focuses on ensuring operational stability in cloud production environments, which is something I'm quite familiar with. I've worked extensively with Azure and AWS internally, assisting numerous customers in building and scaling their services in a sustainable manner on the cloud. This has been a core aspect of my work for the past nine to ten years, and I have a plethora of thoughts and excitement to share on this topic.

Slide 4

The first aspect we need to address is answering the fundamental question: How can we ensure operational stability in a cloud production environment? It may seem like the logical starting point for an engineer is to dive into the technical challenges and solutions. However, I firmly believe that this is the wrong approach. Instead, I propose a trio of key considerations, with technology taking a secondary role.

First and foremost, we must define our risk tolerance. What exactly does risk tolerance mean? It's the extent to which we are willing to make trade-offs and how we articulate these trade-offs concerning stability and operational excellence in relation to our business objectives. The business and its goals must take precedence in this discussion. Engineers must collaborate closely with the business to establish these parameters because they will shape our strategy and technology choices.

To determine our risk tolerance, we can start with easily quantifiable factors, such as availability levels. What level of uptime do we expect? Are our competitors consistently achieving five nines of uptime? If so, the market may dictate that we aim for the same. Can we go above and beyond? Or is our service oriented towards casual users, allowing for some downtime? Conversely, if our service is critical to enterprise customers or a significant revenue source, maintaining high uptime becomes paramount.

We must also consider the broader context within our organization. Is our project merely a front-end addition with minimal reliance? Or is it a core component that, if disrupted, would affect the entire organization? Understanding usage patterns, cost implications, and potential revenue increases tied to improved uptime is crucial. This information guides our decisions and trade-offs.

In my experience at AWS, I was once tasked with designing a system for 100% uptime. While we could have achieved it technically, certain business constraints made us settle for a lower but acceptable uptime. Such discussions between engineering and the business are common and necessary.

Additionally, we need to account for external factors and dependencies. Are there third-party services or other companies relying on our uptime? The ripple effects of our downtime could send signals throughout the ecosystem. We must assess and plan for such scenarios.

In summary, we begin by defining our business needs and risk tolerance. Next, we adopt a DevOps methodology as the right cultural approach to building operationally stable systems. DevOps, in this context, refers to the culture and practices of engineering teams taking responsibility for the entire lifecycle of their services, from development to maintenance and even handling incidents. This culture encourages quality, accountability, and continuous improvement. DevOps practices include "we build it, we run it," where engineers deploy and maintain their services, leading to better code quality and faster feedback loops. Another practice is conducting blameless post-incident reviews, focusing on understanding the root causes of issues and sharing lessons learned across the organization.

Additionally, we establish risk budgets, granting teams the authority to deploy within certain risk thresholds. This empowers teams to make decisions about their service's stability while aligning with organizational goals.

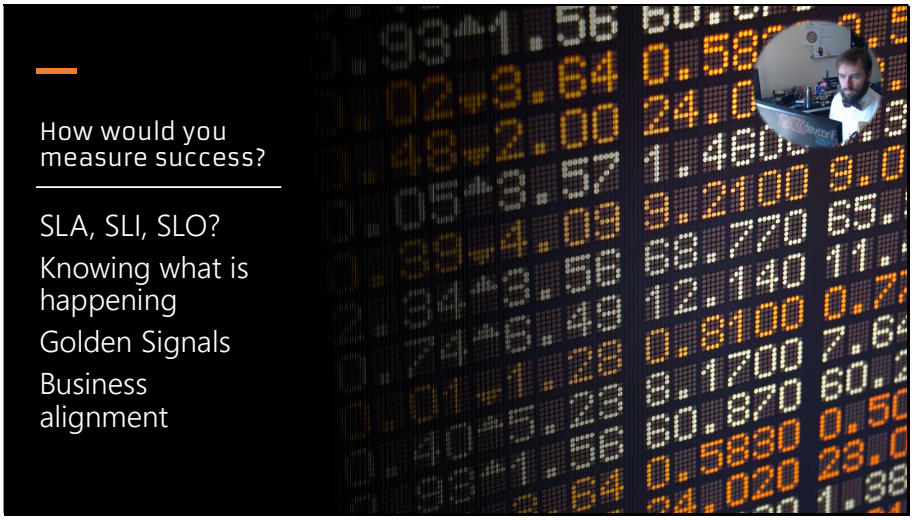
Finally, we consider the technological aspects of ensuring operational stability. This involves designing fault-tolerant systems, understanding scaling patterns, and being mindful of the cloud environment's limitations and opportunities. Scaling decisions must align with business objectives, ensuring cost-effectiveness and efficient resource utilization.

Furthermore, we should explore multi-region and multi-cloud solutions to avoid vendor lock-in and potential cost spikes. Understanding the cloud ecosystem is essential, especially for those transitioning from traditional on-premises environments.

In conclusion, a holistic approach that balances business needs, culture, and technology awareness is key to ensuring operational stability in cloud

production environments. This trio of considerations forms the foundation for building robust and resilient systems.

Slide 5



Now that we've discussed the aspects of operational stability, let's focus on how we measure success in this endeavor. To do this, I like to introduce three essential acronyms, starting with SLAs (Service Level Agreements). While SLAs are commonly used when dealing with customers, they might not work as effectively at a team level. Instead, I prefer to start with SLIs (Service Level Indicators). SLIs are simple, specific metrics that are universally understood across the business.

For example, we can define an SLI called "errors," which quantifies how many errors are acceptable per month. It's crucial that when we mention "error" to one team, it means the same as when we discuss it with another team. This establishes a common language, which we document and demonstrate how teams can measure these indicators. SLIs can cover various aspects such as outages, user numbers, or data volume. Some SLIs may stand alone, while others might need to be connected with other SLIs to provide meaningful insights. Ultimately, these indicators roll up into Service Level Objectives (SLOs), which are defined by the business. SLOs specify what the business expects in terms of error rates, uptime, user scalability, and so forth. We separate the responsibility for SLIs (engineering) and SLOs (business) while working towards both objectives.

The next critical step is to ensure that we are aware of what's happening. Having SLIs is valuable, but we need tools like Grafana, Kibana, or other

modern observability tools to surface this information. This means incorporating engineering efforts into our software development. Dashboards should be built, maintained, and reviewed regularly by the teams themselves. Teams are more likely to care about their SLIs when they have visibility into them. Alerts should also be configured for thresholds to prompt action when necessary. Fortunately, there are user-friendly tools to facilitate this process.
A common question is, "Where should I start when setting up an SLI or a dashboard?" The answer lies in what we call the "golden signals," which are four critical metrics to monitor:

- Latency: Measures the time it takes for a specific action to occur.
- Traffic: Examines the volume of users, requests, or interactions with the service.
- Errors: Tracks the number of errors occurring in the system.
- Saturation: Monitors how well the underlying systems handle the workload, which can include CPU and memory usage.

These golden signals serve as a foundational starting point. Some of them may be composed of multiple SLIs, while others may be single indicators.
Lastly, it's crucial to align all these measurements back to the business. SLOs help align business objectives, but it's essential to connect this with our risk tolerance and risk budgets. Measuring success from an engineering standpoint is valuable, but it becomes truly meaningful when it aligns with and makes sense to the business. Engaging in these conversations allows us to explain trends and improvements, demonstrating the value of our efforts to the business stakeholders.

Slide 6

The next section that I was asked to talk on today was around handling technical debt.

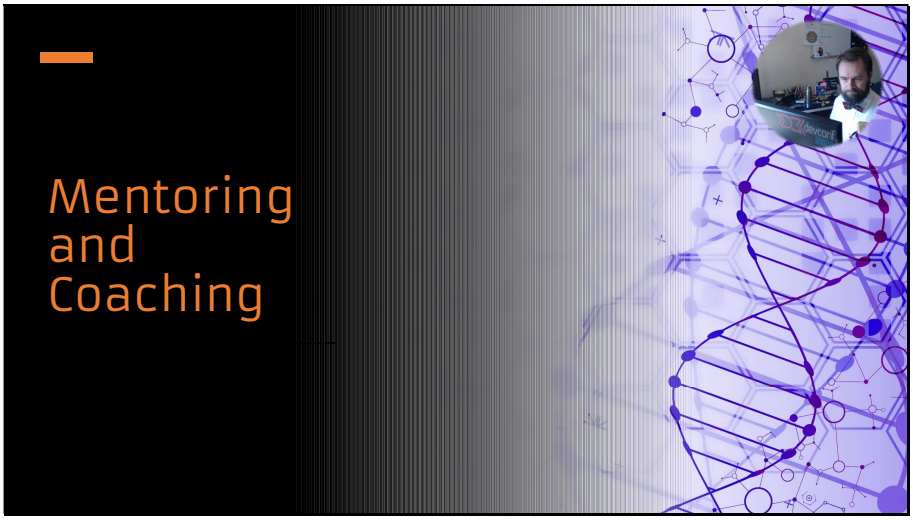
Slide 7

And the first question I got asked is, can you give a pitch to an executive team about why we should reduce technical debt?And I thought this is a fascinating question, and I'm probably the person who's going to give you the most diverse answer here, because I'm not sure we should always, but we'll get to that in a moment. But let's start off with what is technical debt. And technical debt is ultimately the outcome of making trade offs away from building the perfect piece of, perfect system, perfect piece of code.Because ultimately, that's what we do as engineers. A large amount of our work is to understand and build those, and document those trade offs. What could a trade off be? We may have a hard deadline. an event's coming up, start of a school season, we need to get some features out for that. We may choose to have our set of principles that says we have, we have 90% code coverage and we need lots of testing and we want our code to be very good and we use Sonolint to check these things and we decide, you know, we're not going to make this deadline, we will cut some corners.And we make that trade off. And then the thing we had traded away is that debt and that debt could be bad code. It could be lack of test coverage. It could be a number of things around that. And it is often intentional, even if it's not vocalized. So a lot of the time engineers will, particularly earlier on in their career, will make trade offs themselves.Oh, I think I can do it this way with this. And they kind of build a system and they don't think of all the trade offs. And you can have technical debt that way. You can have technical debt as well from

just poor engineering effort. Again, the more we talk, and we'll talk about later on, we get to code reviews and mentoring, but there are things we can do prevent that.avoidable debt from happening. The real concern in that case for us is often technical debt that is made intentionally. Those trade offs that we do make to achieve our goals. And so this does beg the question, is all of this bad? Should we actually just go and reduce all our technical debts? And my answer to that is, it depends.I don't think that all technical debt is bad. I think some technical debt is technical debt. I don't believe perfect systems exist. And so you will have some trade offs that have led you to a certain path. I often find though that engineers, particularly new tech leads, and people come to a team to look at old codebases and they go, oh, this is bad, and this is bad, and we must fix this up, and this is...wrong. And it's not that they are incorrect because they're absolutely correct that that stuff was probably wrong, but they don't understand the context it was built in. And so this is why for me, when I do documentation, I often follow the Moscow pattern where we document our musts, shoulds, coulds, but importantly, our won'ts.The things that we intentionally are trading off, going back to, you know, when we're building a feature, why did we not add enough unit tests for this? Well, we should have a piece in our doc that says. We, we decided to not pull a lot of unit tests in this because we didn't have the time. And we decided as a team that we would come back to it in three sprints.That might be something you put in a doc. And now when somebody comes in a while later and they look at this and go, Oh, we don't have this. Oh, here's why we didn't have that. And they can reason through it. So I think there is some of that can feel bad, but not actually as bad. Ultimately what is really bad technical debt?It is things that prevent us from achieving our business goals. That's all bad technical debt. Good technical debt is like, the code might be a little messier, or we haven't quite had our code coverage numbers, but it's not actually impacting anything. If we chose to trade off writing tests to go foster, And we find that that scenario doesn't actually impact us in future.It's not bad. It's debt, but it's not bad. If we find that our deployments after that are failing more often because we don't have good test coverage, that's a bad technical debt. And that's the type we should fix. We absolutely should do that because it's impacting our ability to ship, to deliver for the business.Lastly, how do we tackle that? Are there different approaches? And there are. We can do Big Bang. I've seen teams do technical debt sprints. Really useful. I often do with my teams. Hey, let's go and flag this as technical debt that we know, and we are going to set a deadline to do that. resolve it. And if it's not resolved by then, then we will have a big bang approach.But I tend to lean more to incremental, cleaning up as you go. We often talk about in my teams about the Boy Scout rule and how you should

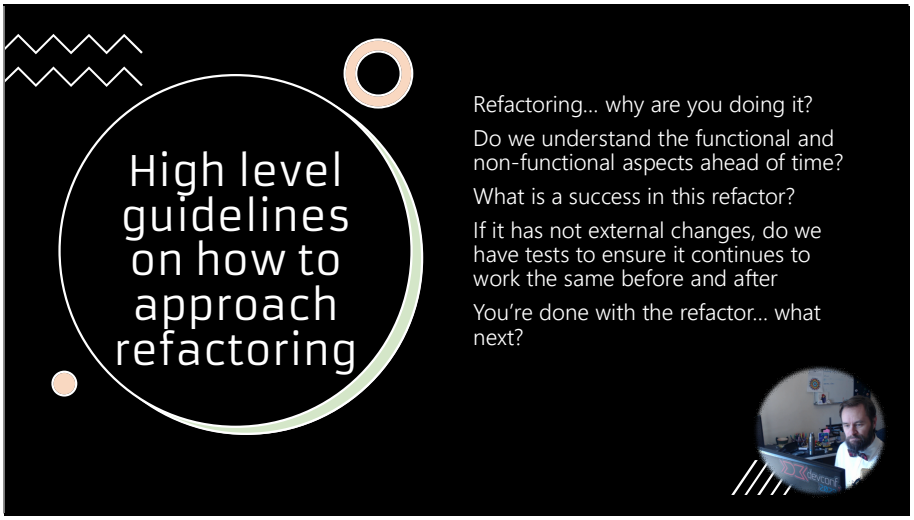
leave the code better than you find it. And that encourages teams to make small fixes. while they're doing other work. And that small fixes reduces that technical debt.So I tend to lean more to incremental. I think it gives you a better outcome over a long time, but there are scenarios you need a big bang. This is something I deeply have dealt with many times and deeply passionate about. I have a few hundred word article though. If you are interested in more depth on this, that's available there and how you can approach it.

Slide 8



Moving on to mentoring and coaching and another area I'm very passionate about because I have worked with and running internships, I have worked in communities, I run the largest conference for software developers. Mentoring and coaching is core to my business. I'm very happy to talk about this.

Slide 9



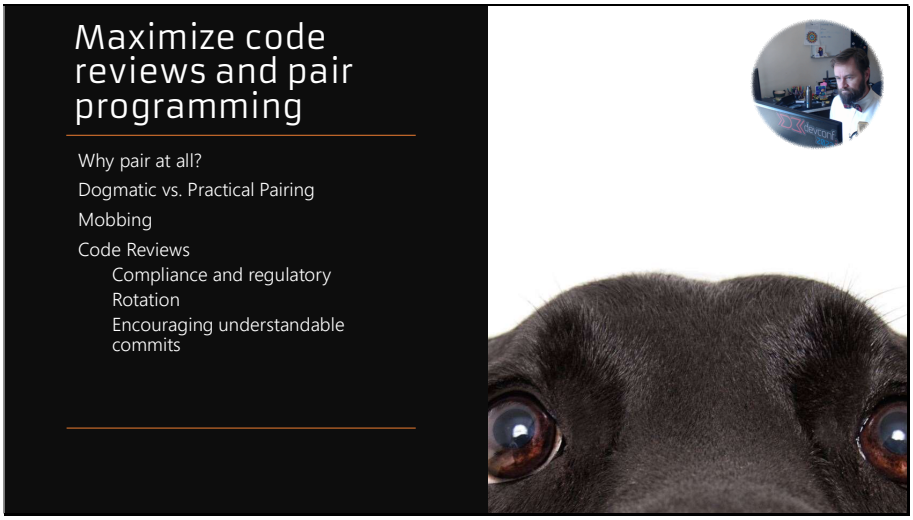
We begin with a fundamental question: What advice can we offer to a team of engineers with mixed skill levels on how to refactor code? To answer this, we must first ask why we want to refactor code. It's a question similar to the one we posed regarding technical debt. Why are we assuming that refactoring is necessary? Often, engineers embark on refactoring projects without a clear understanding of their objectives. Are we refactoring because we attended a conference and were inspired by a speaker's new approach? That might not be a sound reason. Are we aiming to enhance performance, improve code quality, or enhance readability and comprehensibility? These are valid reasons, but we must always start with the "why." Understanding why we're undertaking a refactoring project will shape our approach to it.

Once we've established the "why," we need to delve into the functional and non-functional aspects of the task. For example, if our goal is to improve performance, we must determine the functional requirements. These requirements might include specifying how much faster the code should run, how it should handle various scenarios, and any non-functional requirements, such as accommodating specific user levels or completing the refactoring within a set timeframe. Understanding these aspects is crucial for selecting the appropriate steps to achieve our goals.

Additionally, we must define what success looks like for this refactoring endeavor. This is where having unit tests and test coverage in place becomes

essential. Suppose we're refactoring internal code that won't alter the external experience but will enhance internal processes, as in the case of performance improvements. In that case, we need to ensure that our changes maintain the same input-output behavior. This may necessitate revisiting unit tests, integration tests, API tests, or even implementing load tests for performance enhancements. Without this testing, we cannot confidently determine the success or worthiness of our refactoring efforts. Therefore, we might need to address testing infrastructure before proceeding with refactoring.
Furthermore, if our refactoring doesn't result in external changes, we must verify whether we have testing measures in place to ensure that the code functions correctly before and after the refactoring. After we've completed the refactoring process, we're not done. We should play it back, which involves seeking feedback through code reviews and pair programming. We must also identify why the original code was poorly written. Was it due to technical debt, changing business requirements, or other factors? Do we need to initiate broader discussions within the organization? For example, if we found that the code was too slow to handle increased loads, should we communicate this to the broader team and discuss the need for changes? Refactoring often provides valuable lessons that should be documented and shared. This could take the form of a brief presentation during an all-hands meeting or written documentation. In any case, refactoring is more than just code changes; it's an opportunity to build deeper knowledge and share it widely.

Slide 10



The next question I have is, how do we maximize the value from code reviews and pair programming? The question posed to me suggested that these activities are considered informal mentorship, but I find that characterization somewhat surprising. I believe these practices should be integral to our team's operations rather than informal. Let's delve into this topic.

Firstly, let's address why pairing is essential. Why should we involve more than one person in code reviews and pair programming? I will use the terms "Pair" and "Code Reviews" interchangeably in this context. Ultimately, the primary goal is to reduce risk. When only one person reviews a piece of code, it introduces more significant risk than when two pairs of eyes examine it. While this approach may seem slower and potentially more costly, it leads to better code quality, improved outcomes, and knowledge sharing.

To illustrate, I once worked in a team where I was the sole engineer for a few months during the startup phase. However, this setup posed significant risks. For example, when I had to take a few days off due to illness, the team experienced disruptions. Pairing helps mitigate such risks by distributing knowledge within the team.

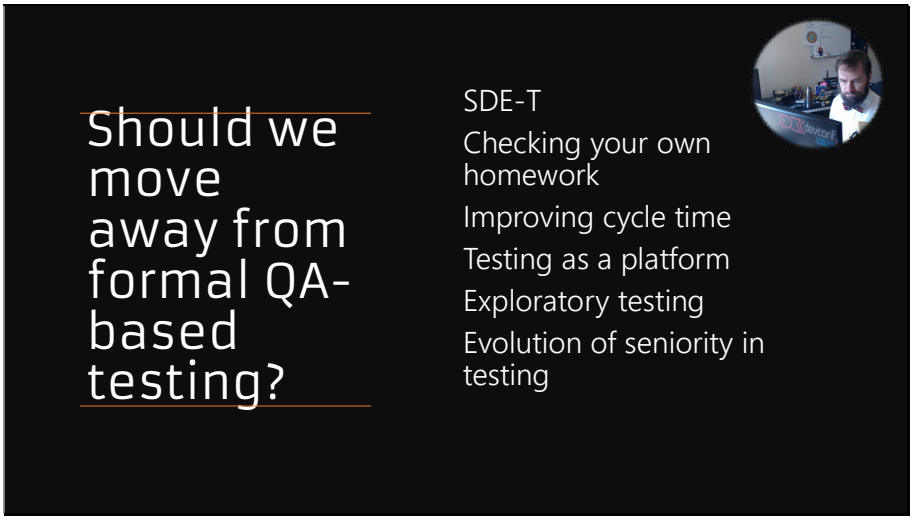
Pairing is also an excellent method for training, especially when you have a mix of senior and junior team members. It provides an opportunity for them to collaborate and learn from each other. Furthermore, it fosters a sense of

culture within the team. Pairing or code reviews can bring remote or isolated team members together through shared activities, improving communication.
It's important to note that there is a spectrum when it comes to the formality of these practices. On one end, there's a dogmatic approach where all development tasks involve pairing or code reviews rigidly. On the other end, there's a more practical approach where you use these practices when needed. The balance depends on your team's composition and needs. For instance, if your team consists primarily of senior engineers who can work independently and require less oversight, you might not need as much pairing. However, code reviews remain essential, potentially even for legal compliance.
In practice, pairing might be a more limited activity, while code reviews could be more standardized. To bridge the gap between them, consider introducing mobbing. Mobbing is a more modern and mature approach compared to pairing, involving the entire team in a collaborative exercise. It has its own set of ceremonies and has proven valuable over the years. Mobbing ensures that not only engineering aspects but also business and user experience considerations are thoroughly addressed.
In summary, the key takeaway is that we shouldn't approach these practices as either dogmatic or entirely informal. Instead, consider a spectrum that ranges from pairing and code reviews to mobbing, and adapt the level of formality to your team's specific needs and circumstances. Code reviews, in particular, offer numerous benefits, such as compliance, psychological safety, promoting understandable commits, and encouraging smaller, more manageable tasks. Embrace these practices strategically to maximize their value and impact.

Slide 11

The next section here, as we get near the end of this, is on testing and code quality. And I've touched on a little bit of that throughout this presentation already. And so we're going to look at just a couple of, well, in particular, we're going to look at one particular aspect in addition to what we've already spoken about.

Slide 12



This is a fascinating topic, as the question presented suggests a prevailing trend in software development teams: the shift away from formal QA-based testing in favor of developer testing. This trend has a long history, dating back to the 1980s when we at Microsoft referred to it as "SDTs" or Software Development Engineers in Test. In many places where I've worked, testers are developers with a specific testing focus. However, I've also encountered organizations, often more traditional and enterprise-oriented, where QA engineers exclusively handle testing. In my view, this separation isn't necessarily a sound strategy, but let's explore this further.

One common concern driving the reliance on dedicated QA is the fear that developers, who write the code, might be checking their own work, which appears risky. However, when we consider the practice of pairing, where multiple engineers collaborate on code, this risk is significantly mitigated. Additionally, with robust tooling integrated into pipelines, such as quality gates, unit tests, code coverage checks, and tools like SonarLint that identify specific patterns, it becomes challenging not to maintain a high level of quality, regardless of whether you have dedicated QA or not.

In fact, I tend to favor having engineers write their own tests. The emphasis here is on automated testing because manual testing doesn't scale efficiently as systems grow in complexity. It also leads to testing fatigue, where testers, after repeatedly executing the same tests, may make assumptions. Thus,

concerns about engineers checking their own work must be balanced against the drawbacks of having dedicated QA, including fatigue and scalability. The solution to these challenges is automation—automate as much as possible. This involves implementing automated unit tests, integration tests, API tests, and schema tests, among other strategies.

By incorporating automation, you not only improve testing rigor but also experience faster development cycles. You eliminate the need to wait for a code submission to be tested by QA, and this reduces the time and cost associated with bug fixes and code improvements. However, does this mean that QA is no longer needed? I believe that QA remains a crucial element of software teams today. However, the role of QA should evolve from performing manual tests to building a testing platform.

In this context, I envision QA teams developing the testing infrastructure and tools that engineers can use. For instance, QA can focus on setting up automated API tests or exploring new ways to enhance automation. They establish the platform, and developers plug into it. This approach streamlines the development process, resulting in quicker testing cycles and greater efficiency.

Another essential aspect where testers continue to play a vital role is exploratory testing. This form of testing, where a tester explores a UI without specific guidance to discover issues, remains challenging for automation. While AI might offer solutions in the future, the human element in exploratory testing, with its ability to think creatively and simulate real-world user behavior, remains irreplaceable.

In summary, the shift away from traditional QA roles in software development teams is not so much a trend as it is an evolution in the role of QA and testing practices. Over the years, testing has matured, testers have gained experience, and testing culture has evolved. As a result, we've seen the emergence of testing as a platform, where QA teams build the tools and infrastructure for developers to conduct automated testing efficiently. This evolution enhances the force multiplier effect of QA within a team, allowing them to contribute more significantly by empowering engineers to embrace automated testing, rather than focusing on manual testing and bug tracking.

Slide 13

So with that I thank you so much for your time. This has been a 40 minute talk, so hopefully it's something you enjoyed and I hope to hear from you very soon.