The reliability paradox: Why less can be more

You've made the change from on-premise to the cloud, and your application is running like a charm. In true DevOps fashion you are focusing on building and running the app so you've taken certain precautions: retry mechanisms, fast failovers and smart alerting rules have been implemented. While the resilience of the system is improved, we should avoid the mindset that we are completely in control of the system's reliability.

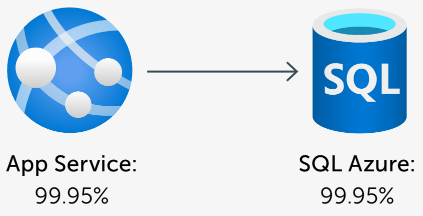
Authors Geert van der Cruijsen and Casper Dijkstra

When we ask customers how reliable their application should be, expectations usually are around 100%. That would be desirable indeed, but is this really a target worth pursuing? Which price are we willing to pay for overly high availability targets? In order to answer this question, we should get some insight into the pros and cons of tightening and loosening reliability objectives. Are you focusing on the right things? Who decides how reliable your application should be? And is there a drawback to too much reliability?

What is reliable software?

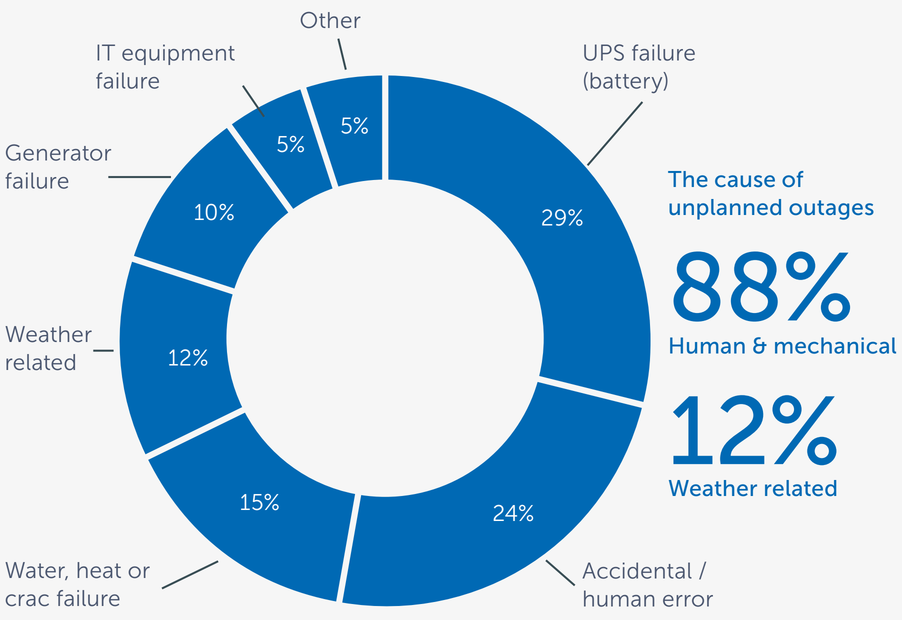
Modern applications are based on multiple cloud components. These typically come with a Service Level Agreement (SLA) of three nines (99.9%) or three and a half nines (99.95%). Let's focus on the interplay of Azure App Service and an underlying SQL database as an example.

Both services have a guaranteed uptime of 99.95, so around 21 minutes of downtime are allowed per month. Our application needs both services to behave correctly in order to be fully reliable.



Since these components are independent of each other, the App Service can be down on Monday from 06:00 to 06:20 and Azure SQL database can be down the ensuing day from 14:20 to 14:40. Because the services can have outages at different times, the compound SLA of multiple components is of course lower than their individual targets. Where they both satisfy their own reliability target, the overarching application may have a lower availability.

Then, all application components are communicating through the network of which we know that it is not always reliable. Starting to think of it, there are a lot of mechanical or human errors or natural disasters that may incur entire data centers outages. This means that our systems have an inherent risk of unavailability which has to be (and usually is) endorsed by developers, the business and its end-users. We should therefore expect that each modern application exhibits some degree of unreliability. But this does not have to stress us out. We will see that the impact of these (often short-lived) outages is smaller than commonly thought.



(https://www.365datacenters.com/portfolio-items/overcoming-causes-data-center-outages/)

There are architecture patterns that could be used to minimize user impact on certain issues.

Should we start paying significantly more on data redundancy offerings (like Geo-zone-redundant storages) to reduce our unreliability to an absolute minimum? We think that this should always be a business decision focusing on the business impact of certain failures.

Certain failures in our application can occur without the user being impacted, how important are these issues? We should not aim for perfection, but find the correlation between unreliability and user satisfaction. To find out, we should dive into the impact of failures – when does it actually matter?

Embracing the risk of failures

Aiming for higher reliability targets may seem like a reasonable (and ambitious) goal to pursue for product owners. We want to convey that setting higher targets is not always the right thing to do, and there may be high and concealed costs. If we want to improve the reliability of our system from 99.9% to 99.95%, and the application generates an annual revenue of €500.000, then a reasonable estimation of the additional revenue is only €250.

Moreover, there are many scenarios in which small unreliabilities do not bring about any noticeable consequences. When your LinkedIn feed is rendered incredibly slow, you probably press F5 and the problem is already over. There are many scenarios where neither economical nor user satisfaction factors run the risk of being drastically reduced. Let's focus on a warehouse example involving availability!

Example scenario: Warehouse solution

You're building software to handle all incoming orders that need to be collected in the warehouse by robots. This process is a key process within your business, so it should never be interrupted. If the robots stop working, trucks can't leave on time and customers won't be happy because their packages are late. So how reliable should things be? Our robots should never run out of work. This is a good business impact that we could measure. But what happens when communication to the robots fails?

Communication to the robots is super important, so our initial thoughts might be that we should do everything in our power to make this super reliable, but what if the robot can store up to 10 orders in advance? If each order takes about 30 seconds to complete, you have 5 full minutes before a robot runs out of work. So when looking at reliability, we should aim for a solution that focuses on achieving this business result instead of solely measuring which percentage of the messages to the robots were sent successfully.

It goes without saying that end-users care about reliability. However, we should form a realistic picture about which expectations customers have in mind about the application. When the effect of enhancing the reliability from three to four nines (99.9 to 99.99 percent) gets obfuscated by the unreliability of external factors (causing extra reliability to go *unnoticed*), then we can reasonably be reluctant to improve the reliability. Spending time on either rapid new feature development, lower latency or reducing accumulated technical debt would have been more fruitful for our end-users. The key things to monitor should be focused on user and business impact rather than technical errors.

Full reliability is overachieving, a single database failure is catastrophic and this uncertainty leads to imminent stress among your employees. There are ways to improve the reliability, but this should always be a conversation between the business owner and the engineering teams that build and manage the application.

Defining objectives that customers care about

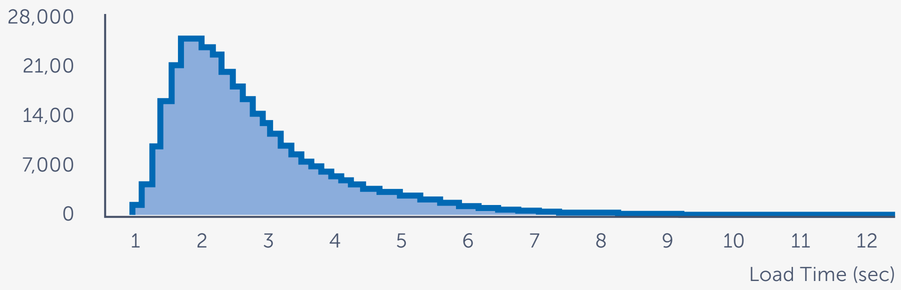
Instead of focusing on overly high reliability targets, we should use our experience and common sense to contemplate which level of service we want to provide to our customers. The well-known service *level agreements* (SLAs) are backed up by *service level indicators* (SLIs) and *objectives* (SLOs).

While any measurable quantity can be promoted to an indicator, we recommend choosing just a *few good probes*. These should encapsulate what users deem important in the application and it's usually a good idea to start working backwards from customer experience to SLIs rather than setting objectives based on accumulated data. Google advocates that the most useful indicators of your system's health are: Latency, Traffic, Errors and Saturation, which they've coined *golden signals of monitoring*. What these indicators have in common is that all of them pertain to internal structures of your application.

The trick is to not lose ourselves in the anomalies of our internal system, but to get involved in the translation process to our end-users. For instance, the errors indicator does not always map directly on the user experience, but it's a fair bet that the following Service Level Indicators are strongly correlated with user satisfaction:

- *latency* (nobody wants to wait 2 seconds for each HTTP request);
- *availability* (2% downtime is simply too much);
- *throughput* (it shouldn't take too long to upload pictures);
- *correctness* (your shopping cart should show your selected items).

Now it's time to form objectives for these indicators to trace how much unreliability can reasonably be tolerated, and unsurprisingly, we'll look at the impact on customers. A frequently made mistake is to form objectives based on averages. This is problematic because distributions of our indicators are usually right-skewed, where the first 1% of the users have slightly better behavior, and the last 1% have incredibly slow responses of multiple seconds.



(Source: www.lognormal.com)

The risk of a far-reaching right tail is a valid concern, and in our experience it is useful to create objectives for high percentiles (e.g. 95%, 99% or even 99.9%) rather than for averages. This is based on the line of reasoning that every user has a good experience when the worst-case scenarios have reasonable experiences.

While the indicators are usually the same for different systems, objectives are where variation comes into play. Our objectives are reflected in the SLA and this sets expectations for the customers, and they may or may not select our service based on them.

Customers have different expectations about user-facing systems (webshops, social media) compared to archiving systems and, to name another, big data processing systems like Apache Spark.

SLI	SLO	SLA
Latency	95% of requests should be served within 100ms	within 150ms
	99% of requests should be served within 300ms	within 400ms
Availability	99.95% uptime per month	99.9% uptime per month

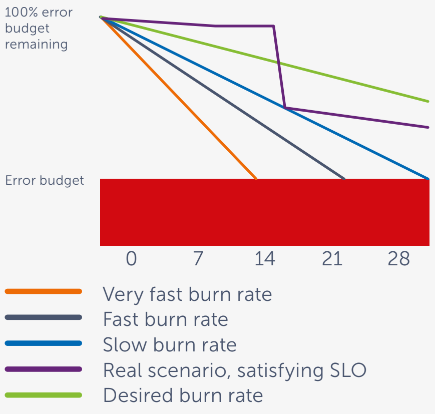
While end-users certainly care about these indicators, there is not always a trivial mapping function of these indicators to customer experience. What would provide more insight is to look at this through a functional angle, i.e., how do the application users experience the application? Spending excessive amounts of time on optimizing the reliability of the system means less time for rapid feature innovation, automation and experimentation.

A good example would be Outlook versus Youtube. The first application is used by businesses throughout the world for communication, and they expect the service to have a high availability. Youtube on the other hand is not used for critical purposes. While users may be somewhat annoyed by a lower uptime, this is probably outweighed by the positive experience of rapid bug fixes and new features. Google has also set lower reliability targets on Youtube versus Gmail for similar reasons.

At this point we should have a feeling for setting reasonable SLOs. What should we do next? The next step is to find a perfect balance between reliability and innovation!

Finding the sweet spot between reliability and innovation

Error budgets are based on the idea that a certain amount of unreliability is acceptable. For instance, Azure SQL strives for an availability target of 99.95%, which means that they are permitted to have a downtime of 21.6 minutes per month. These 21.6 minutes constitute their *monthly error budget*. Where traditionally outages would have been stressful events in need of immediate investigation, modern Site Reliability Engineering principles state that everything is under control as long as the error budget has not been burnt. Likewise, we can (and should!) form error budgets for microservices that we maintain.



This illustration shows the acceptable burn-rate (the blue line) and two unacceptably high burn rates (+25% and +50% slopes). On the other hand, the green line denotes a more positive trend: a burn-rate at which we would easily satisfy our objective. Now let's take a look at our real error budget burning - the purple line. During the first two weeks we're gradually spending a little bit of the error budget.

Then, due to a Daylight Saving bug on day 14, an excessive amount of error budget is spent, stopping at our *acceptable burn-down rate*. This means that we have to be more careful at this point, and we decide to reduce our release velocity. After a week (day 21) we notice that we're doing way better than the blue line, we can actually *embrace more* risk again.

Without an error budget, the business would have probably thought that we're not delivering enough value - the system was unreliable for quite a while! The change of philosophy with an error budget has noticeable advantages. Engineering teams can safely deploy and stay focused on what they were doing, and nobody is alerted when a fraction of the error budget is scorched. More intriguingly, we may even claim that most of the error budget should be used and this provides an excellent opportunity to experiment. Nowadays, it is even considered a best practice among Site Reliability Engineers to not aim for a significantly higher availability than our target, since this creates false expectations for the future.

The error budget reminds us that unreliability is not always undesirable. In fact, it even provides a *minimal amount of monthly downtime* (of course lower than the SLO target). If we haven't burnt any error budget, we simply haven't taken enough risks and customers will start to rely on their experience that the system is always reliable (which means they will be more bothered by future issues…)

Actionable metrics as a conversation between business & engineering

What we find a great benefit of Service Level Objectives and Error Budgets is that these create realistic expectations about the system that engineers and the product owner have agreed upon. Moreover, it removes a great deal of subjectivity out of any conversation on the application's health: we know exactly how much failure is permitted while keeping the end-users satisfied with the product

Aiming for overly high reliability targets has another issue: it is at odds with the desire for new features. Feature development is pretty dangerous from a reliability point of view:

> the complexity of the product increases with each new feature;
> in fact, each code change comes with implicit risks and changes the (assumed reliable) state of production.

While development teams are evaluated on their feature development velocity, tension can arise between business and engineering teams. An error budget is a very effective means to establish a balance between reliability and innovation. When the error budget is on track, we should not hesitate to develop and deploy. The system behaves as expected, and our end-users are certainly happy when new features become available quickly!

When we're on track for our objectives, we should feel encouraged to experiment. The risk of incurred unreliability is outweighed by the value provided to users by our experiments (or to the development team by automating recurring tasks).

Let's take a look at new technologies. These often have the potential of adding a lot of value to the application, but they involve a risk for the reliability of the system. Let's give some concrete examples:

Potentially interesting experiment/improvement	Could provide value	Risk for reliability
More rapid feature development	– Users can get preview features faster	– Bugs are more easily introduced – Rollbacks
Migration from ARM to newer Infrastructure-as-Code frameworks	– Easier code changes – Better maintainability – Unit tests can be written for the infrastructure	– Not sure whether first deployments are successful – Downtime
Chaos engineering on production	– Expose vulnerabilities in the system – Rigorously test alerting scheme that is in place – Better understanding of strong and weak spots of our application reliability	– Chaos is invoked for a subset of the users – Reliability will certainly be lower than without chaos engineering tool

Being encouraged to experiment, we know when we are allowed to embrace risk for the greater good!

Getting started with reliability engineering in your application!

We have shown examples of SLOs raising expectations about the system's functioning and that loosening SLOs can have advantages. Error budgets help us to assess how much time we can spend on innovation versus improving reliability. Expecting full reliability comes at undesirable costs, of which we highlighted:

> Increased stress among employees:
- When the system is not behaving perfectly (even when nobody is using the application at that time).
- Resistance against feature development - what if something breaks on production?

> Customer expectations:
- Usually not as high as commonly thought.
- They tolerate occasional hiccups in the system.

> The tradeoff between reliability and innovation:
- The business and end-users not only care about reliability, but also about other aspects.
- Setting reasonable reliability targets allows us to make smart tradeoffs.
- Being on track for these targets means that we can embrace risks and experiment (to use new technologies, automate things, deploy faster et cetera), all of which may increase productivity and user satisfaction.

We hope that this article helps as a conversation starter for many organizations in order to make engineers and business work together in terms of thinking about reliability and how they can work together on making the right decisions for building an application that is as reliable as required. </>

Geert van der Cruijsen
Digital Kickstarter, Enabler for companies to embrace DevOps, Cloud & improve their engineering culture

xpirit.com/geert

Casper Dijkstra
Cloud Engineer

xpirit.com/casper