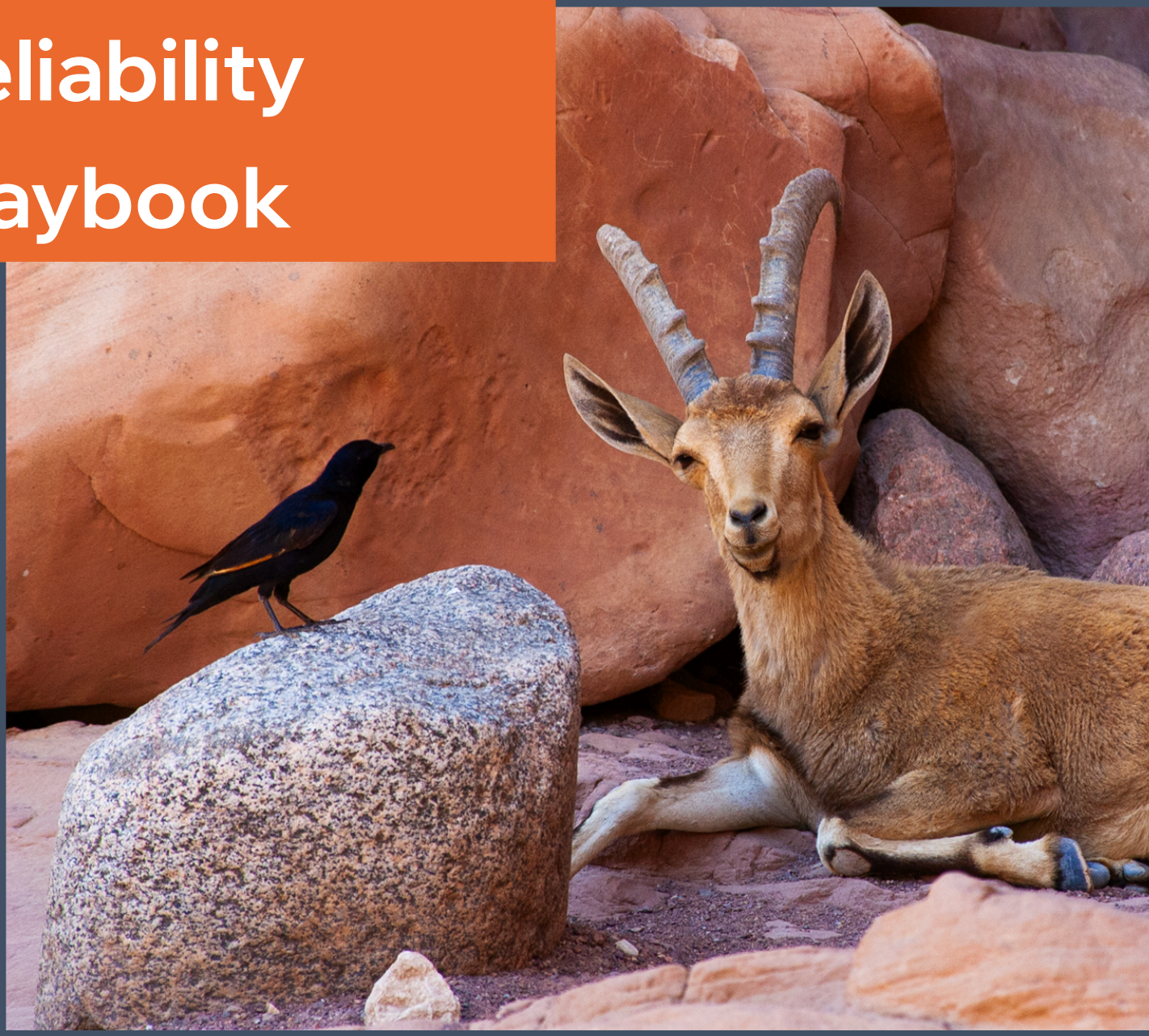
Microservices Reliability Playbook



Written by

Yoav Abrahami

WIX Engineering

Contents

Contents **2**
Introduction **3**
Risks in Software **5**
Malfunctions 5
Security 6
Change 7
Load and Latency 9
Introducing Microservices **11**
Service Level Objectives **12**
Measuring Reliability 13
Predicting Reliability **14**
Example of Predicted Reliability 15
Monitoring Microservices **17**
Recommended Alerts Strategy 17
Reliability with microservices **19**
The ideal reliable system 19
Microservice Patterns **20**
Read patterns 20
Write Patterns 28
Multi-Service Patterns 32
Call Patterns 36
Summary **42**
About the Author **43**

"When you have a monolith, you've got one big problem to solve. Switch to microservices, and now you've got 99 smaller problems—plus a distributed system".

Maintaining reliability with microservices distributed systems are challenging. This playbook explores the risks involved, the theory why, how to measure, alert and predict reliability and how to mitigate using architectural patterns.

Introduction

I first learned about microservices in 1998. Yes, a quick search will reveal microservices as a name was introduced in 2011-12, attributed to **Martin Fowler** and **James Lewis**. Yet, in 1998 I read a book by **James R. Callan** about collaborative computing from which I learned all about microservices, 13 years before they were named.

I joined Wix in 2010 to figure out what the iTab is and how the company's product can work on mobile. However, by the end of the year I have transitioned to join the team building Wix Infrastructure with **Eugene Olshenbaum**, **Aviran Mordo** and the rest of Wix Engineering.

With this work, we will explore why we should be using microservices, how large they should be, what are the risk factors when using microservices, how to mitigate those risks, how to monitor microservices and guidelines to build reliable microservices systems.

To understand microservices, we first need to understand how they evolved. In the 90s, we have been using what is now called 2 tier applications - a desktop application connected to a central database (it was before the time of web or mobile applications).

Those desktop applications had to contain all the functionality, from all the UI to all the business logic, and in some cases database migration logic (for lazy migrations). Developing those applications was very productive using tools like Visual Basic, Delphi and later C#.

However, as the application grows and the number of developers grows to over 10, the effects of working in a monolith emerge - different teams are not ready at the same time, efforts for merging work, different assumptions on the database, etc.

Late 90s and the early 2000s, we moved to 3-tier applications, placing a server between the desktop application and the database. This server decoupled the business logic from the UI, allowing for two teams to work in parallel. Yet, with application and complexity growth, it was clear we need a more robust pattern.

From the mid to late 2000s, the shift to web applications created modular client applications, as each web page is independent. Microservices emerged at the same time to decouple backend development and enable scaling the team, as well as create smaller blast radii in case of a problem.

But what is the underlying cause for those shifts and what are we solving?

Risk!

Risks in Software

Risk in software is defined as the chance that given a deployed software system will fail to function, due to any reason. There are 4 types of risks for software systems

Malfunctions

Malfunctions happen when some infrastructure breaks, such as the hardware, network, electricity, the cloud provider has an outage or similar. Obviously, if such a thing happens, our software system will fail to function.

Some Software systems are designed to be resilient to malfunctions (most notably IP networks) by deploying adaptive routing algorithms and making a design decision of no global true state.

In most cases, when building software systems we assume our infrastructure is reliable, or we handle malfunctions using availability zones or multiple regions (for cloud applications) and multiple data centers (for on prem systems) and switching from one to the other in case of malfunction.

Security

Risk from security incidents is the risk of someone, employee or a 3rd party, going into the system and preventing it from working, partially or as a whole. Ransomware attacks are just one such example.

Another type of security risk is data breaches, at which someone uses a software or access vulnerability to steal data or proprietary information.

Malicious users can also abuse a system, using it for unintended intents such as attacking other systems, impersonating, consuming unintended resources, etc.

Normally, security teams deploy different measures to prevent those types of risks, which we will not expand on in this playbook.

Change

The risk of change is the risk caused when a team of developers deploy a software change. With the change, on every deployment, they have a chance to

1. Introduce a bug
2. Introduce a breaking change some other component is not ready for (dependencies)
3. Introduce additional CPU / IO / memory load the deployment is not ready for (application load)
4. Broken deployment
5. Adversely affect other components in other systems which may seem unrelated to the original deployment (also denoted as blast radii of the failure)

This is the type of risk microservices aim to solve.

We can calculate the reliability due to risk of change of a component using the following formula:

$$1 - \frac{\left(\begin{array}{c}\text{time to detect}\\ \text{and fix a}\\ \text{problem}\end{array}\right) \times \left(\begin{array}{c}\text{percent of}\\ \text{deployments}\\ \text{with a problem}\end{array}\right) \times \left(\begin{array}{c}\text{blast}\\ \text{radii}\end{array}\right)}{\left(\begin{array}{c}\text{period of time between}\\ \text{deployments}\end{array}\right)}$$

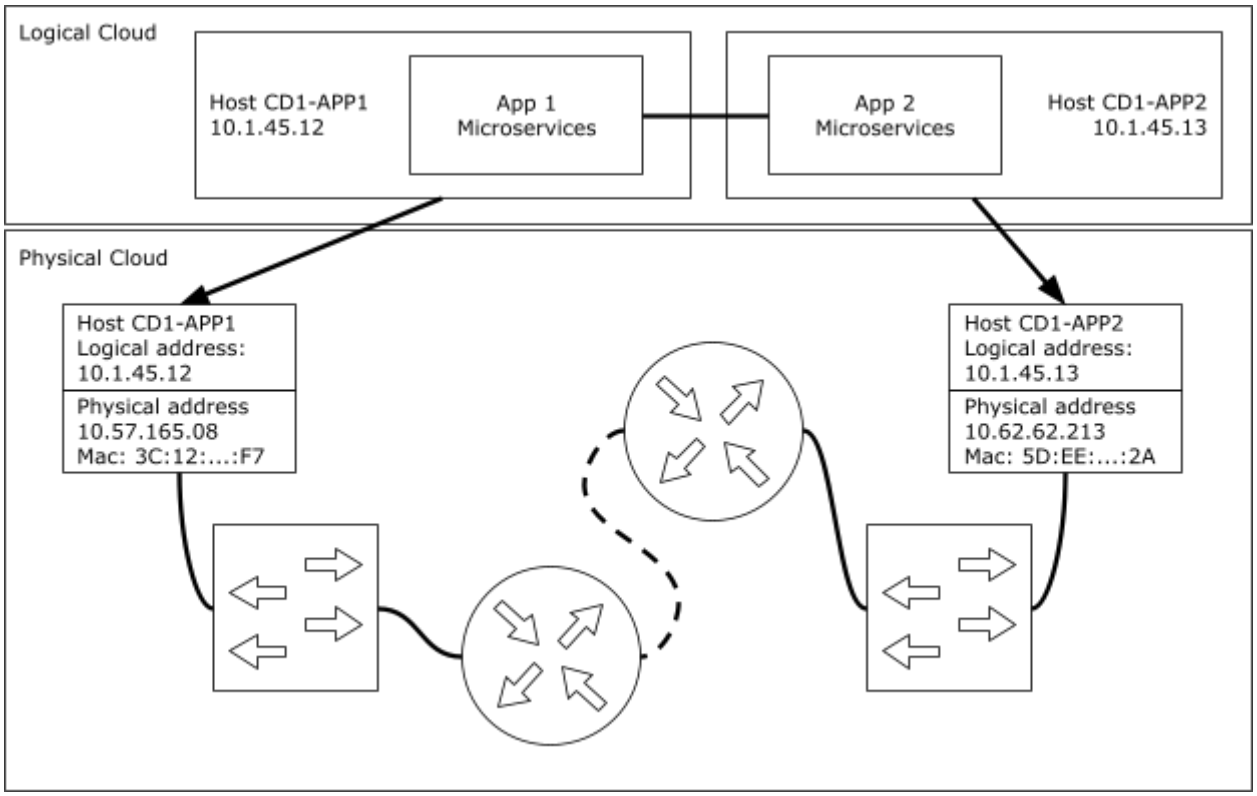
1. “Percent of deployments with a problem” - can be mitigated using methodologies like TDD (test driven development), lab deployments and manual QA.
2. “Blast radii” - is given a problem, how much of the overall system is affected?
3. “Time to detect and fix a problem” - can be mitigated using modern monitoring and alerting tools as well as having fast automated deployment pipeline

Microservices aims to reduce the “blast radius” of a problem. Continuous Delivery aims to reduce the size of a change and those reduce the “percent of deployments with a problem” as well as the “blast radius”. Overall, both reduce the risk of change.

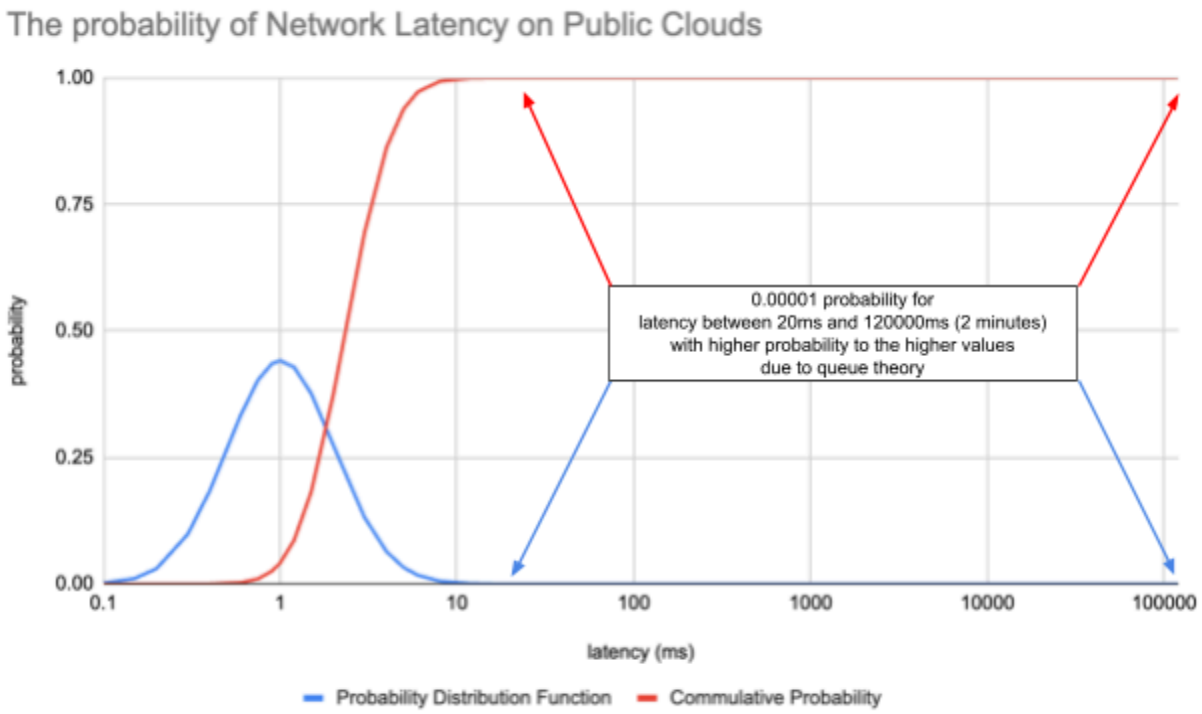
Load and Latency

The risk from load and latency is the risk that given two processes (two microservices, server and database, client and server, or any other two), the call from the first to the second will delay up to a timeout due to different loads and latencies in the places we do not see.

To understand load and latency, one just needs to consider what happens between two adjacent cloud hosts - which appears virtually to be side by side. The reality is they are probably on different locations in the cloud data center, and the packets get translated from the virtual network IP address to the physical network IP address, goes through a switch, a few routers, a switch again, target network stack and process. All the devices and layers deploy queues for load management, which most of the time offer quick response, yet sometimes offer really slow response.



The result is similar to a Log-Normal distribution with a cutoff after which higher latency numbers have higher probability due to how queues are working. In this work we use a model based on Log-Normal distribution (μ=0.5, σ=0.7) with a cutoff at 20ms, which effectively gives five-nines network (latency probability of 0.99999 to be below 20ms). The probability is visualized below.



To summarize load and latency risk, two micro services communicate at a five-nines reliability, but will fail to communicate at 0.00001 probability.

Introducing Microservices

Microservices is an architectural style that breaks down an application into small, independent services that communicate with each other using well-defined APIs. These services are designed to be loosely coupled, meaning they can be developed, deployed, and scaled independently. In essence, a microservices architecture is about building applications as a collection of small, independent services that work together to achieve a larger goal.

The first question being asked about micro services is how large should they be? A single function? A single file? A persistent entity and its APIs? A logical Module?

The simplest answer is that a microservice is as large as the team managing it. One team can own multiple micro services, but one micro service can be owned by only one team.

To understand this statement, one just reviews the risk of change above. If a microservice gets contributions from two or more teams, in order to release the microservice both teams have to synchronize on changes to the microservices and have all changes in a state of done. There is an added risk that one team deploys (for example, due to a hot fix) while another team change is not done, and by deploying introducing a bug.

But is that the only rule for microservices size? Actually no, there is another rule that will cause a team to have multiple microservices - managing blast radius, or the SLO (service level objectives) of the system.

Service Level Objectives

Before diving into Reliability of microservices, let's review how we define service level objectives for a software component. SLO is composed of 4 definitions

1. Availability - the ability of a system to give a response, including an error response. A client calling a system and receiving a timeout (from the system, or the client decides to stop waiting), for that client the system is not available.
2. Latency - the time it takes to get a response, including an error response. Do note that latency turns into an availability problem once the client or some intermediate communication element decides to stop waiting.
3. Error rate - when the system gives an error that is not the fault of the client. We are talking about HTTP 500s type of errors. To be clear - HTTP 400s errors, such as a client accessing unauthorized API and getting an error should not be counted as part of this error rate - this error rate captures the inability of a system to fulfill a valid request
4. Correctness - the ability of a system to give a correct response. For example, we expect the add API to return 2 for 1+1. If it returns any other number, the response is not correct. Normally, when a system is not able to return a correct response, we say it has a bug.

All the above should be measured on a per request basis, not on a time basis. The reason is that software systems do not have the same load at all times, and using time basis gives more weight to the low load periods. Take that to the extreme - a system that is up and running idle 24x7, yet gets 10 calls in one minute and fails all of them, is it 99% available (according to time basis) or 0% available (according to request basis)?

For this discussion, we define **reliability as SLO availability * (1 - error rate)** and measure it with the x-nines measure, or percent of operations that succeed. Five-nines means that 99.999% of the operations succeed.

Measuring Reliability

As stated in the previous section, we define reliability as

$$(\text{reliability}) = (\text{availability}) \times (1 - (\text{error rate}))$$

Reliability is not a global attribute of the system - instead, it is an attribute of each independent API of the system. A system with 50 APIs has 50 reliability scores, one for each API.

Side note - the above statement depends on how we define an API. If we have an API “do anything”, which performs multiple different actions based on different inputs, it is not very beneficial to measure its reliability as one API. In such a case, we can measure reliability per API and key API inputs which determine which action to perform. Consider an API such as run task, which gets a task id as an input - as different tasks can be quite different, it makes sense to compute reliability per task and not just per the API.

To measure reliability, one has to monitor each API for availability (when it is called and does not answer in a reasonable time) and error rate (when it answers with uncalled for error).

For example, an API with 10,000 calls during a day, of which

- 40 calls have failed to reach the API
- 100 calls have reached the API and timed out
- 200 calls returned a 5XX error

The API reliability is then 1 - 340/10000 = 1 - 0.034 = 9.964%.

Predicting Reliability

Reliability of a distributed system can be predicted based on only two measures of the system itself, which factor in the risks of Load and Latency and the risk of Change. Those two measures enable architects, developers and managers to predict the reliability of a system, or the impact of a change to the system on the system reliability.

The risks of security incidents and malfunctions depend on external elements and cannot be predicted from examining the distributed system itself.

The two measures are

1. **Number of Network hops**: The number of network calls between services, up to data persistence.
2. **Number of Artifacts**: The number of releasable software artifacts involved.

Reliability can be predicted using the two above measures by

$$\left(\begin{matrix}\text{Predicted}\\ \text{reliability}\end{matrix}\right) = \left(\begin{matrix}\text{Load and}\\ \text{latency}\\ \text{reliability}\end{matrix}\right)^{\left(\begin{matrix}\text{Number of}\\ \text{network hops}\end{matrix}\right)} \times \left(\begin{matrix}\text{Change}\\ \text{reliability}\end{matrix}\right)^{\left(\begin{matrix}\text{Number of}\\ \text{artifacts}\end{matrix}\right)}$$

Given we can assume the single network call reliability is five-nines or 0.99999%, and that Change Reliability per artifact is also five-nines, or 0.99999%, the model formula simplifies to

$$\left(\begin{matrix}\text{Predicted}\\ \text{reliability}\end{matrix}\right) = \left(0.99999\right)^{\left(\begin{matrix}\text{Number of}\\ \text{network hops}\end{matrix}\right)} \times \left(0.99999\right)^{\left(\begin{matrix}\text{Number of}\\ \text{artifacts}\end{matrix}\right)}$$

Or simplified to

$$\left(\begin{matrix}\text{Predicted}\\ \text{reliability}\end{matrix}\right) = 0.99999^{\left(\begin{matrix}\text{Number of}\\ \text{network hops}\end{matrix} + \begin{matrix}\text{Number of}\\ \text{artifacts}\end{matrix}\right)}$$

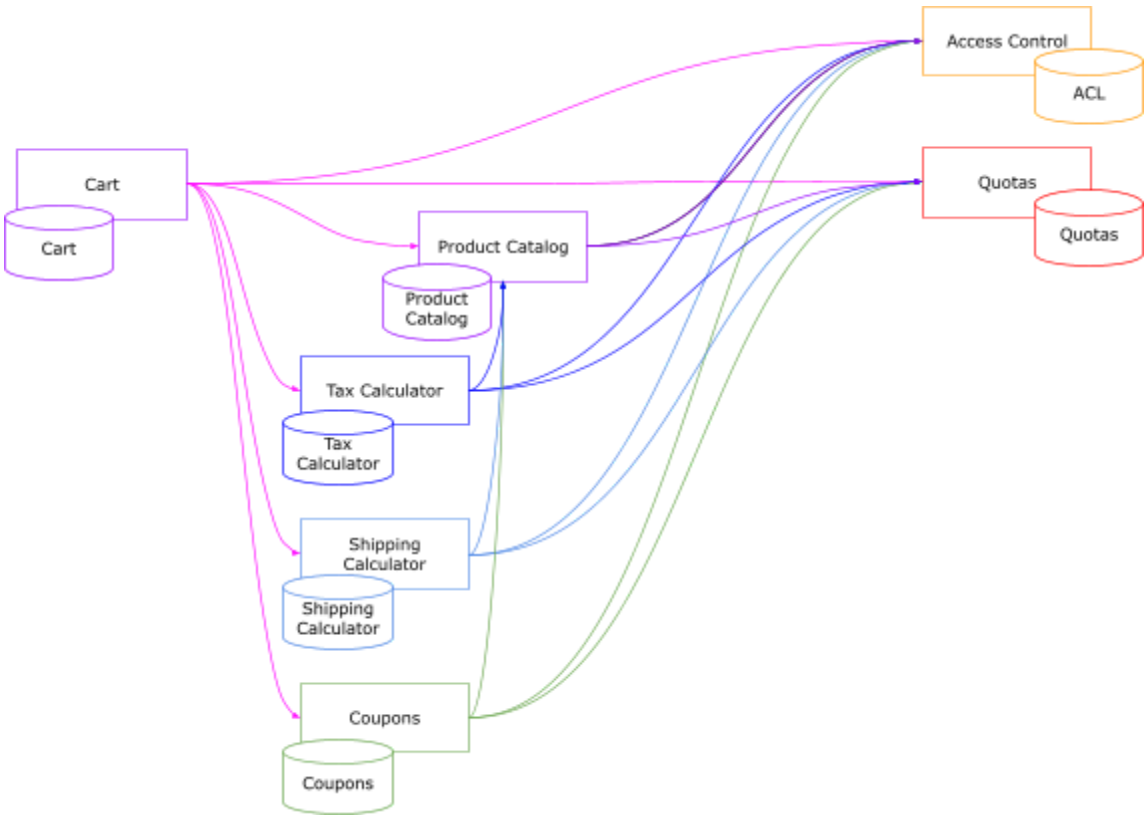
Example of Predicted Reliability

Let's consider a toy model of a commerce cart, that includes multiple services -

1. A cart service, owning the line items and cart totals calculation
2. A product catalog
3. A tax service, which calculates the tax depending on the items in the cart
4. A shipment service, which calculates the tax depending on the items in the cart
5. A coupon service, which validates coupons added to the cart and assess their value and applicability to the cart items

In addition, our micro services system has non functionals for access control and quota.
We also assume each micro service has a database, and each call to a micro service includes one query to the database.

Our system looks kind of like the following:



And now, let's examine the transaction to load the cart data for display, including the line items and total price.

The number of artifacts involved is 7 (with micro services we count the database as part of the micro service deployment unit).

The number of network calls is 35

- 1 Cart to Cart DB
- 6 Cart to other micro services + 6 each micro services to their DB
- 8, from 4 micro services to 2 micro services (from Product Catalog, Tax calculator, shipping calculator & coupons to Access Control & Quotas) + 8 for ACL and Quotas DB Access
- 3 additional calls to product catalog from Tax calculator, Shipping Calculator and coupons + 3 calls to product catalog DB.

Using our formula, the predicted reliability of the system is 0.99999^(35+7) = 0.9995 or 99.95%

Monitoring Microservices

Ok, we have defined how to predict the reliability of a system only using the number of network calls and artifacts. But what is the best way to monitor and create alerts on the system such that we know if we have a problem and can react fast?

Our first intuition is to set an alert every minute (or so) at an error rate of 1-5%.

But does it help? Does it guarantee we keep the SLO we aim for, be it 3-9s (99.9%), 4-9s (99.99%) or above? At the same time, does it cause too much unnecessary noise? How do we balance alerting?

Recommended Alerts Strategy

We note there are different kinds of risks which have different error patterns. Risk of change tends to cause short periods of high error rate, which we want to detect early. Risk of load and latency tends to create a constant error rate that can slightly increase or decrease depending on many factors. We want to factor both and still detect risk of change errors fast without false positives.

1. We remind that reliability is defined per API, and as such to be measured per API individually, based on the number of calls to the API.
2. We want to implement, per API, a measure of
    a. Number of calls
    b. Number of errors
    c. Latency p50, p95, p99, p999 (percentails 50, 95, 99, 99.9)
3. We need to define what the latency limit is. Any API call latency above this threshold is considered also an error.
    a. This threshold has to be below the timeout limit, which is considered an error by default.

    b. or semantic threshold at which the API returns a regular response while logging the latency as an error.
    c. For most API, this threshold can be at around x10 of the mean latency, or x2-x3 of the p90 latency.
4. We want a fast alert at a lower threshold, set as
    a. One minute interval for high traffic APIs
    b. Moving average over 5 minutes, computed every minute for lower traffic APIs
    c. With error rate limit of about 5-10%
5. We want a slow alert at a higher threshold, set as
    a. One day interval, or
    b. One week moving average, computed daily
    c. At almost the SLO target limit (for 4-9s, we set error rate at ~0.001%)

Explanation

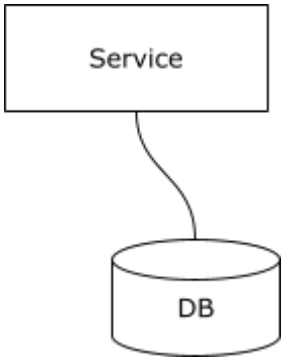
1. The latency threshold is intended to detect momentary load and latency risk effects - if an API latency becomes considerably larger but does not reach the timeout threshold.
2. The fast alert is intended to detect risk of change effects - if some changes cause the service to fail, we want fast feedback.
3. The slow alert is intended to detect load and latency risk effects - gradual degradation over time, or a buildup of errors over time. By setting a threshold that is close to the required SLO we can both detect early and have time to react and fix.

Reliability with microservices

Now that we have a model for predicting reliability, and we have the understanding that reliability is dependent on the number of releasable artifacts and number of network calls, we can define the ideal reliable system

The ideal reliable system

The ideal system, and in fact the only system how meets the five-nines target, is the following



Obviously, the ideal system looks like a monolithic system!

So, are microservices the wrong choice? Actually, no. Microservices mitigate other types of risk, such as risk of change and reducing blast radius in case of a problem. Microservices solve an organizational problem of multiple teams working concurrently and independently.

So the next question is how do we regain reliability with a microservices architecture? Here comes microservices patterns!

Microservice Patterns

We categorize the patterns into 3 groups - subsystem patterns, Read / Write / Process patterns and call patterns

Read patterns

When considering a single micro service, the most common concept is that of CRUD services, which support reading (R) and writing data (CUD). However, we have to keep in mind that services commonly also process data, such that whatever is read from their database or other services is processed before a read operation. The processing can be sync or asynchronous.

For this discussion we define Read / Write / Process (RWP) as a service that supports data written into it, the data can be processed by the same service and by fetching data from services (enrichments) and supports reading the processed and enriched data.

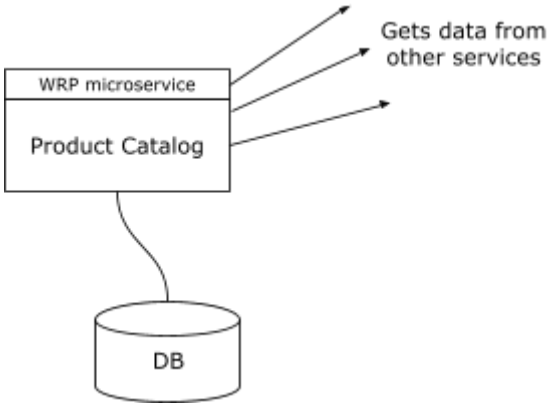
Read patterns differ in the level of decoupling of the read operations from the write and processing operations, to decouple the risk for the read operations from the rest of the system.

The read patterns are summarized in the following table

Name	Optimize for
Single RWP Service	Simplicity
Reader	High reliability, any database schema
Reader with Preprocessing and Enrichments	High reliability, simple database schema, preventing other service calls during reads
Isolated Reader	High reliability, non-simple database schema, enrichments. This is the most reliable and fast reader pattern.
Writer	High reliability and fast consistent writes
Multi-Writer	Higher reliability for write operations
Command Query Responsibility Segregation (CQRS)	High reliability and fast consistent writes & High reliability and fast reads

Single Read / Write / Process (RWP) Service Pattern

The classic micro service



This type of system couples risk of change and risk of load and latency all into one process. A write operation may prevent a read operation due to load, or a change in write or processing logic can cause read operations to fail, or vice versa.

Considering such a system that, for example, during read or write operations, the processing is calling another 10 microservices (transitively), the system reliability drops as per the above formula for predicting reliability.

For instance, with a product catalog service, the additional calls can be to validate a write, enrich the catalog with inventory or other information, categorization, etc.

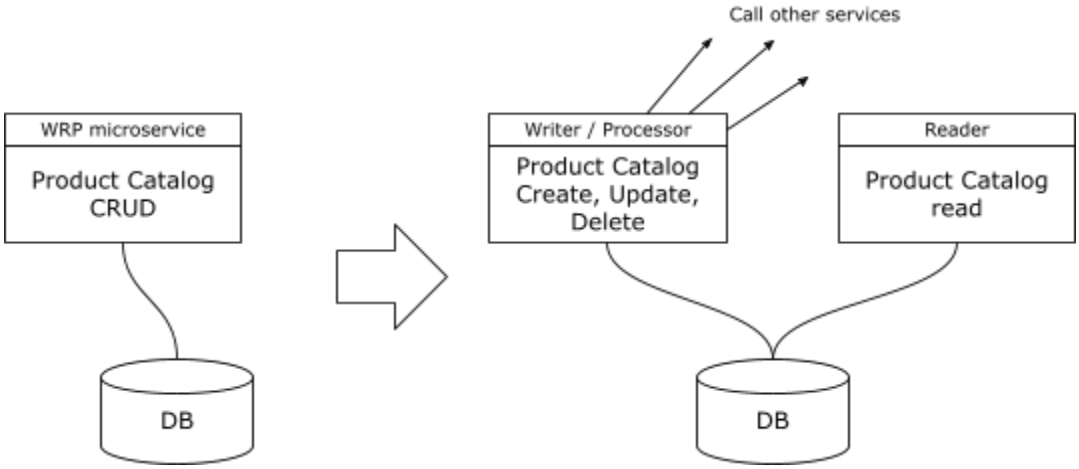
This is the simplest and most straightforward way of building a micro-service and that is a good starting point for talks on increasing reliability.

Reader Pattern

The Reader pattern mandates that we split a system into two microservices (reader and writer), favoring higher SLO for read operations. The Reader service is built to be very simple, with reading data from the database and returning it, working on the same database that the writer writes into. The reader may perform complex queries including joins to read the data.

For instance we can consider a product catalog CRUD service. We can consider the get product and search for product APIs as the more critical APIs that require 0.99999 % reliability, while the rest of the update APIs can have a lower SLO. The product catalog reader service may query the products, categories, variants, inventory and other tables to fulfill the get product and search for product APIs.

The Reader Pattern applied to the product catalog



We also note that regarding risk of change, it is more likely that more changes are applied to the Create / Update / Delete operations of the product catalog, while the Read operation tends to be more stable. In any case, we isolate the read operation from risk of change of the Create / Update / Delete operations.

It is important to note that this pattern is great when the database schema is simple. However, when a read operation requires reading from multiple tables using complex queries, stored procedures or multiple queries, those also affect latency and reliability.

Advantages:

- Simple reader service as an ideal microservice, five-nines reliability.
- Decouple risk of change of write and processing operations from higher SLO read operations
- Isolate high SLO read operations from other lower SLO operations

Disadvantages

- For complex DB schema with references, read operations still need complex queries or multiple queries, which impacts latency and reliability.
- Shared dependency on the database (Reader vs writer / processor)
- Coupling on the DB Schema - changes that require DB change still risk the higher SLO

Reader with Preprocessing and Enrichment Pattern

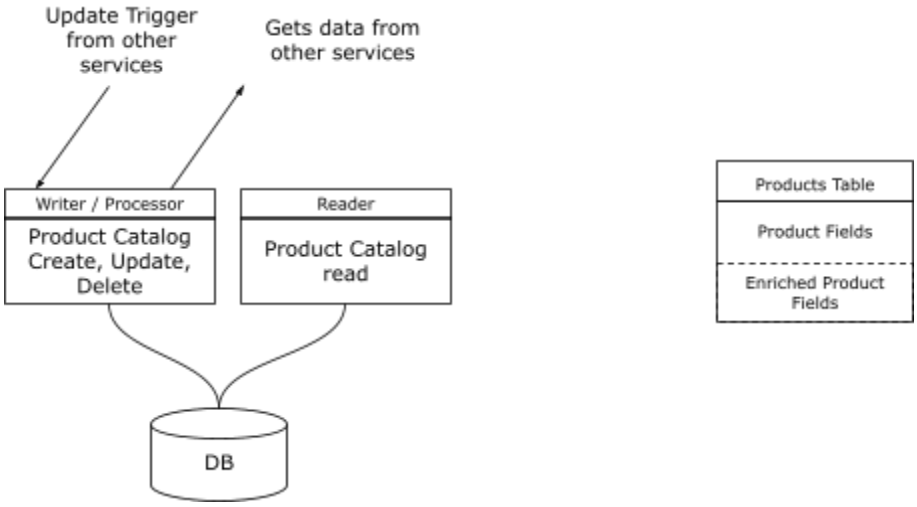
Preprocessing is reading data from multiple tables and creating a new derived table to support read operations. For instance, consider a product catalog preprocessing the products, categories and variants tables into a single product-read table, from which all product information can be read by key.

Enrichments are a type of processing that adds data from other services to the current service data on reads. For example, consider enriching a product with inventory quantity or the association of product with categories and taxonomies (assuming categories and inventory are managed on separate services).

The default RWP service will call the other services on reads to enrich the product information, which means reducing reliability and increasing latency..

The Reader with Preprocessing and Enrichment Pattern mandates that on reads,

1. The service does a simple SQL query to get the data, most commonly by primary key. The writer, on getting an insert or update operation, after updating the normalized table, also updates the derived preprocessed table for the reader.
2. The service does not call any other service. Instead, the enrichments are happening as part of the proceeding on the Writer / Processor service. The Writer, on getting an insert or update operation, calls other services, creates the enriched data and saves it in the database.



One has to ask what happens when the enrichment data changes? For instance, when inventory data changes (due to a purchase or new stock), or due to a change in categories?

First, we note that both inventory and categories for a product are "**slowly changing data**", that is, data that changes a few times a day, at most a few times an hour, but is not interactive data. This allows us to reverse the update flow, and have the inventory and categories systems notify the product catalog Writer / Processor service to re-enrich the products, so that the reader will have them ready when needed.

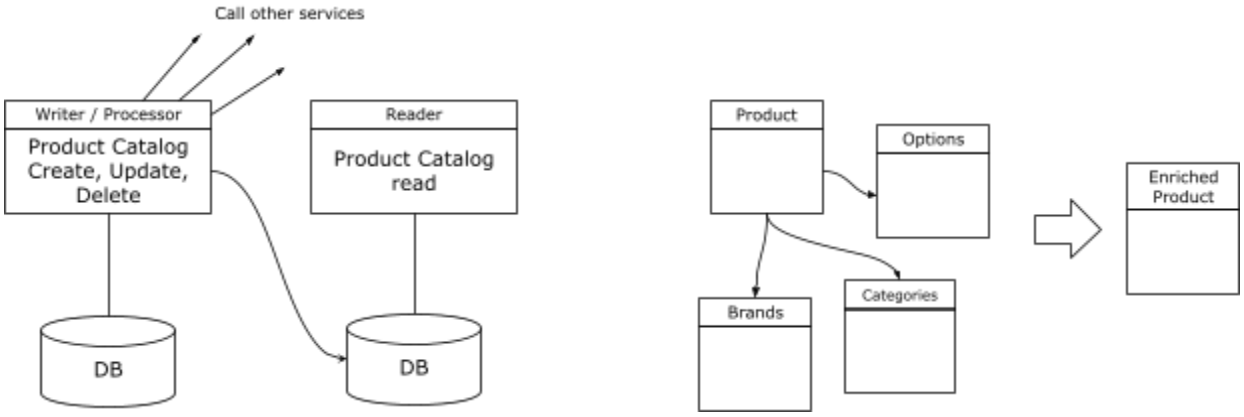
The generic rule is that

1. Preprocessing (without enrichment) applies to eventually consistent data (for which we can regain consistency, see below).
2. Enrichment preprocessing only applies for slowly changing data, such that it is possible to process it beforehand on update triggers (data change events).

Second, did the system lose consistency due to not actively checking on reading the inventory? We claim it did not, as it was not consistent in the first place. Consider the full system includes the client (using a browser or mobile app) who sees the product from the catalog and the availability to purchase. Between the time of seeing the product and adding it to the cart the inventory can change again and again. As a result, the only consistent inventory check has to be done on checkout, on order creation (or reservation logic, if available).

Isolated Reader

The Isolated Reader pattern is a variation of the Read pattern and the Reader with Enrichment pattern to decouple the dependency on the database, decoupling both risk of change and risk of load and latency sourced at the shared database. In addition, the pattern simplifies the database queries on reads to become trivial queries, again improving both reliability and latency.



With the Isolated Reader pattern, the Writer / Processor service and database hold the canonical data structure, while the Reader database holds the enriched, simplified and indexed data prepared for reads.

The Writer / Processor service, on a write / update / delete operation, first writes to the canonical database. Then, it triggers a process to compute the reader database data and update the reader database.

For instance, in the product catalog example, the writer / processor may have tables of products with relations to product options, brands, categories, etc. It can be built as a normalized table with relationships, constraints and whatever makes sense.

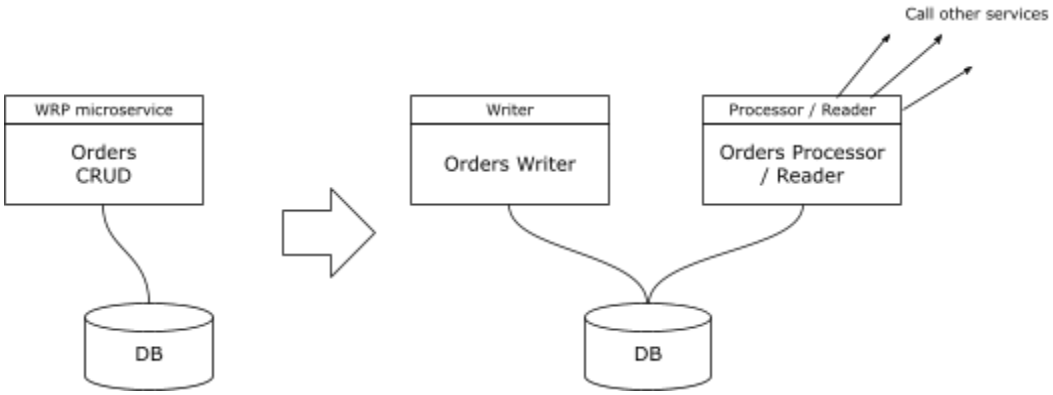
The product catalog Reader products table will be a single table, with all the references materialized, including enrichment from other services such as the inventory data.

The Isolated Reader Pattern extends the Reader with Preprocessing and Enrichment Pattern by isolating the database hardware in addition to the microservices and data schema, decoupling risk of change and load of the database itself. It is the most efficient, reliable and fast read pattern, excluding additional layers such as cache.

Write Patterns

Writer Pattern

The writer pattern isolates write operations from processing. It is useful when write operations have to have highest reliability. The writer pattern assumes a simple write, for complex multi-table or multi-service writes consider the multi-writer pattern.



One example of an application of the writer pattern is the order writer - once an order has been paid, a system has to write the order with a high reliability service to ensure no orders are lost. Then, it can start handling all the aspects of the order, from fulfillment to updating different “my account” pages, analytics, etc.

Once the writing is done, the writer pattern triggers a call to the processor / reader service to process the write operation (e.g. the order).

Advantages:

- Simple writer service as an ideal microservice, five-nines reliability.
- No processing on write - less chance of failure

Disadvantages

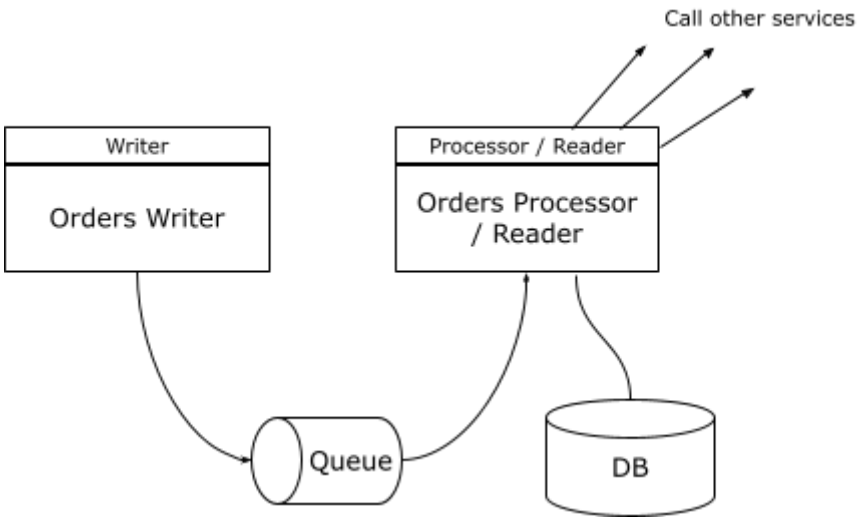
- The call to the processing of the order is non-garanteed, as the service first writes the order, then calls the processing. In case of failure of the processing, we have an order that has not been processed. This requires another mechanism to ensure all orders are processed.

Writer Pattern with a Queue

The writer pattern may use a persistent queue instead of a table for the write, so that both the update of the written data and the processing are driven by the queue. Persistent queues are a great tool to create reliable multi-reaction writes, which can be utilized here.

The writer does only two things - validates the write (access control and business validation) and writes the data to a persistent queue.

This pattern has a downside that it is challenging to get a client to read their own writes immediately, and it is challenging to propagate processing failure to the client. One technique to resolve both is to return a “ticket number” on the write operation, and another API on the processor to get the status of the “ticket number” - was the processing completed and now the writer can read their own writes? Was the processing successful, and if not why?



For instance, applied to the same order service, the order writer writes the order to a queue, which then the processor dequeues, gets the order, writes the order to the database as well as performs any needed processing.

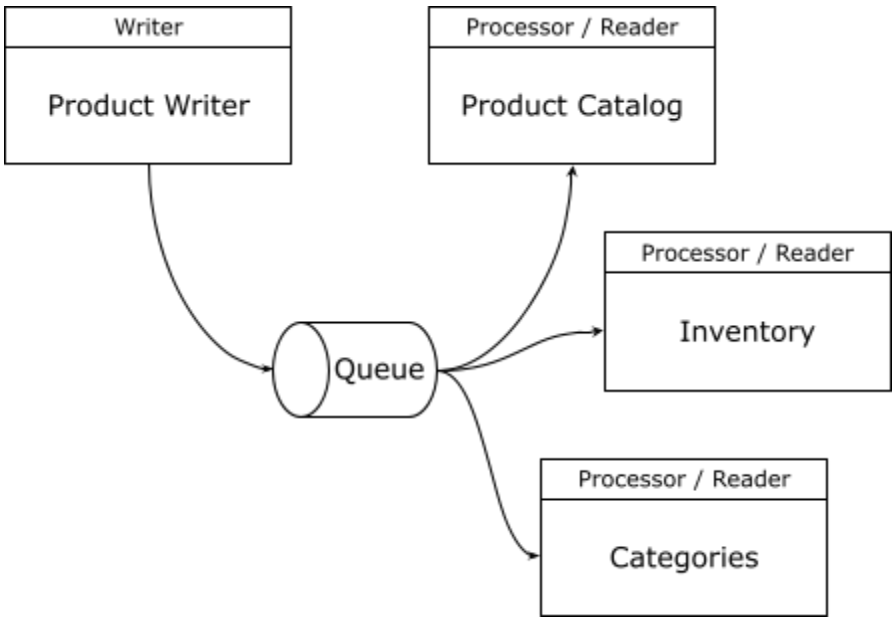
The main advantage of this pattern compared to the Writer pattern is the error handling and persistence we get from the queue. All we need to ensure the data (an order) will be processed is the writer service and the queue to be operational.

In case the processor / reader is not online, or has a problem, the queue saves the written data (orders) to be processed later, or retried.

Also, we note that the writer service is very simple, decoupling the risk of change related to all the processing (order processing).

Multi-Writer Pattern

The multi-writer pattern is a generalization of the Writer Pattern with a Queue, in case we have to write to multiple microservices in a reliable way.



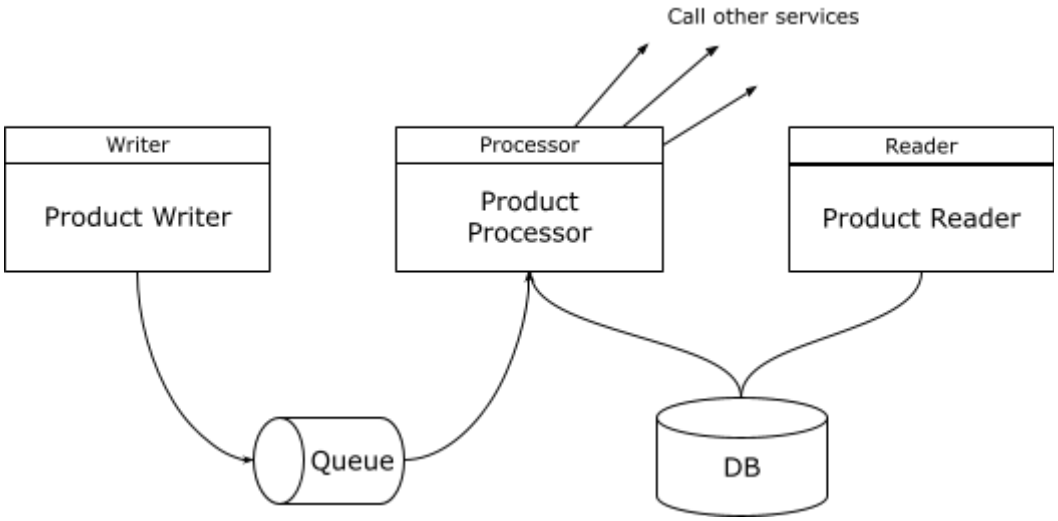
Consider a write operation for the product catalog, at which the catalog, inventory and categories are each managed by a different microservice. The multi-writer pattern allows one to create a single product writer that gets the details of the product, category and inventory, and writes it all to a queue. Each processing service accepts the message from the queue and handles the writes to its database.

The advantage of this pattern is that it decouples writes from processing and reads, while allowing multiple services to be reliably updated.

However, this pattern also introduces a risk that some of the writes have completed successfully, while other writes have failed or are pending due to some delay. The pattern does not guarantee transactional write to multiple microservices, creating the risk of inconsistencies. Such inconsistencies can be mitigated into an eventual consistent system using the persistent queue and a retry mechanism, one that eventually will complete all the writes and restore the system into a consistent state.

Command Query Responsibility Segregation (CQRS) Pattern

The CQRS pattern combines the ideas of the Writer Pattern with a Queue and the Reader Pattern.



The pattern mandates that writes are written as Commands into a queue. A Command is just a message with instructions to mutate data - insert, update or delete, for one or more tables or services. A processor service accepts the Command, does the processing, enrichments and any other operations needed, and writes optimized for reads data to a database used by a reader service.

There are multiple variations of the CQRS pattern, using an event log instead of a queue, deciding when and how to update the reader service (sometimes called materialized views), yet the main principles remain the same - simple writer and simple reader services.

Regarding consistency, what is the source of truth? How does a client see their own writing? How do we ensure no two conflicting writes? All of those topics have multiple solutions,

from optimistic locking to client retrying reads until they get their latest writes - all with advantages and disadvantages which are a whole topic by themselves.

The main advantage of the CQRS pattern is the decoupling of the system into 3 components - the writer which is simple, the reader which is simple, and the processor which can be complex.

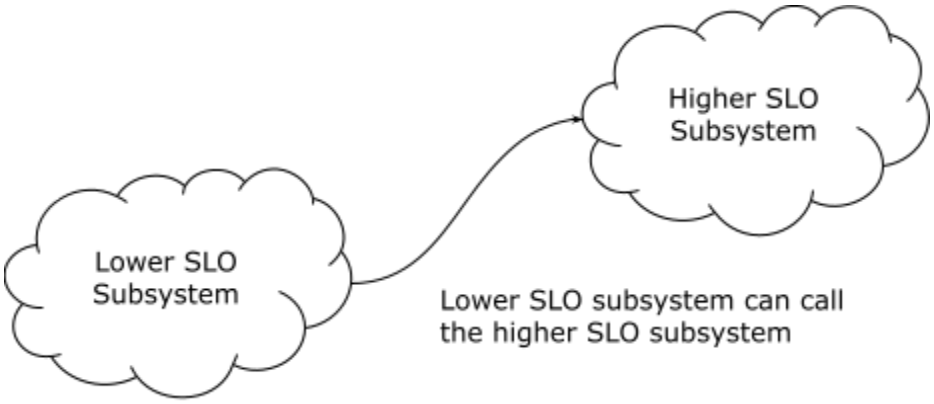
CQRS is frequently used combined with Event Sourcing to create a log of events or commands, the writes. Those events can be partial writes, which are then summarized by the processor into the full up to date system state available to readers.

In addition, this architecture allows creating multiple readers to each get the slice of data optimized for their use case, all based on the same written data.

The pattern optimizes latency for both reads and writes, ensures high reliability of both, at the price of added complexity to the system.

Split by SLO Pattern

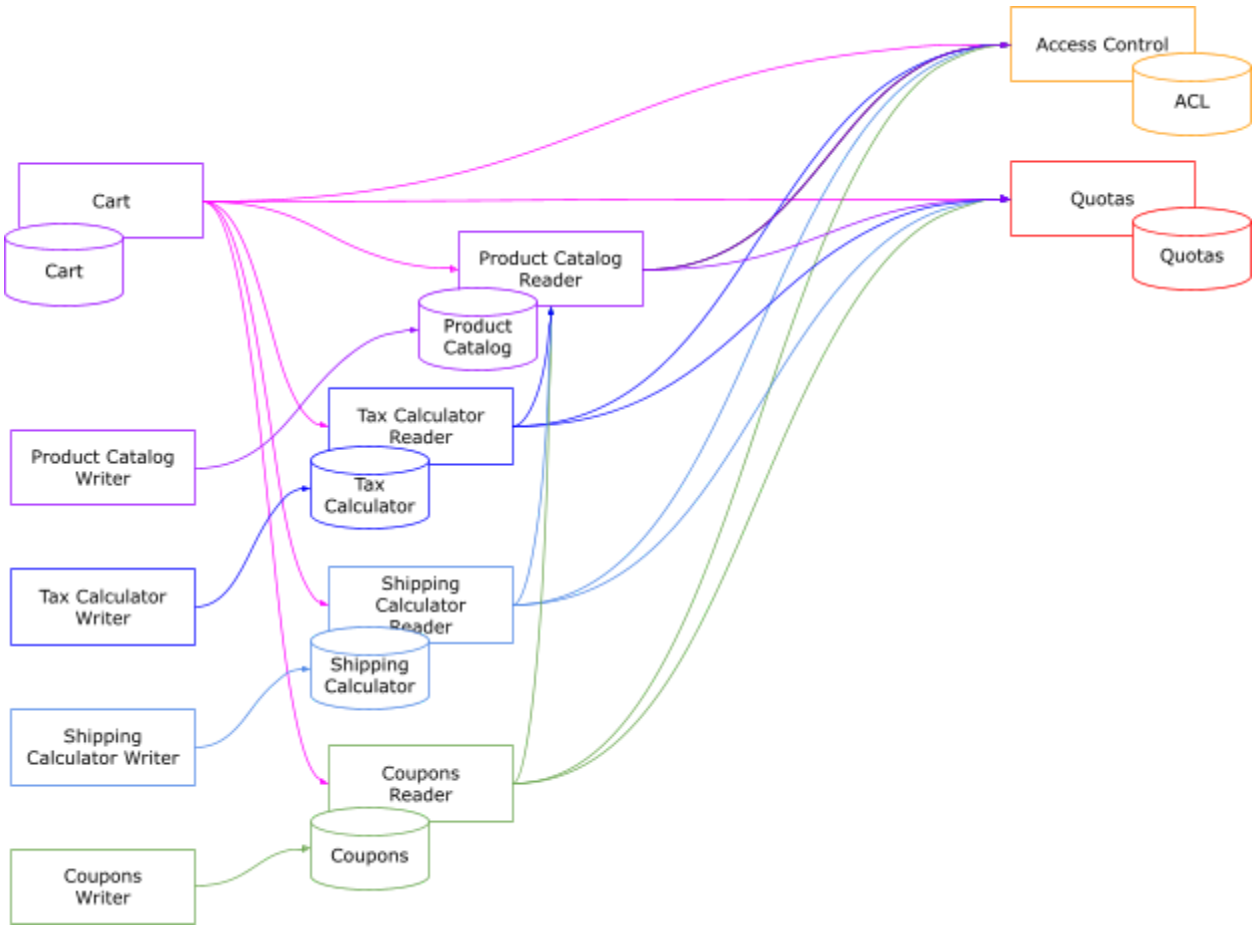
The split by SLP Pattern is similar to the previous (Split Read / Write by SLO Pattern), but on a larger scale. Consider a group of micro services who are supporting multiple business flows. Those different business flows may have different SLA, and as a result we have different SLO operations running on the same set of microservices. The pattern suggests to split the group into two separate groups of microservices, such that one group supports the higher SLO operations independent of the lower SLO group, while the lower SLO group can rely on the higher SLO group.



For instance, consider the cart example we had before. The higher SLO operations are the operations that support the checkout flow, while the lower SLO operations are operations such as updating the product catalog, the shipping configuration or the tax configuration. In addition, analytical operations are also of lower SLO. The split by SLO pattern mandates that we split the system into checkout supporting services and all other services. As we do so, we may employ the Split Read / Write by SLO Pattern for services such as the product catalog, tax and shipment.

Lets focus on the shipping calculator service. The service has one function to calculate the shipping costs of items in a cart. However, it has more functions - to manage the shipping rules probably with CRUD APIs, including validations, maybe simulate calculations for some rules management application, and maybe more operations. We want to decouple the risk of change and risk of load of all other functions from the higher SLO function of calculate

shipping for a cart. We split the service into a reader (who only performs the shipping calculation) and a writer who does everything else, including the CRUD operations of shipping rules.



Note: Why do we consider checkout operations to have higher SLO, while all other operations are of lower SLO? SLO definition is a business definition that considers risk and the ability to absorb delays or other problems vs investment in higher SLO system. For online commerce, the most critical operation is the checkout operation, as it and it alone generates money. All other operations are supporting, and even if they fail, as long as the checkout continues to operate the business is in a good situation.

Call Patterns

Pass Through Cache Pattern

The Pass Through Cache Pattern is using a cache service in front of a microservice. The cache has three functions - improves performance, improves reliability and, in addition, can answer for cached data even if the microservice is not available. Most cache systems are designed to be five-nines, with a single network call to fetch cached data (if not served from memory).

The cache pattern however has a few fundamental problems that limit its ability to improve reliability.

1. Not all microservice API calls can be cached. In a nutshell, we can only cache read operations that are slowly changing.
2. Cache systems are statistical systems, caching only part of the API calls. As a result, in case of a problem with the underlying system, a cache system can only support the cached data.
3. Cache systems are designed to be non persistent. On updates, flush or other events the cache system may have small amount of cached api calls, again limiting it ability to improve reliability
4. A Cache system incurs one additional network call to the cache system, which is required regardless of cache hit or miss.
5. All the above amounts to a challenge to comprehend why some operations are cached and some are not, creating a potentially unexpected pattern of errors for the upstream system calling the cache.

As a result, to calculate the predicted reliability of a system with cache, we need to compute the predicted reliability of the system with a cache hit and without a cache hit, factoring in the additional network call to the cache system, and combine using weighted average the predicted reliability. The formula for predicted reliability becomes

$$\left(\begin{matrix}\text{Predicted}\\ \text{reliability}\end{matrix}\right) = 1 - \left[\begin{matrix}\left(1 - \begin{matrix}\text{Predicted}\\ \text{Reliability with}\\ \text{Cache}\end{matrix}\right) \times \left(\begin{matrix}\text{Cache}\\ \text{Hit Rate}\end{matrix}\right) + \\ \left(1 - \begin{matrix}\text{Predicted}\\ \text{Reliability}\\ \text{without Cache}\end{matrix}\right) \times \left(1 - \begin{matrix}\text{Cache}\\ \text{Hit Rate}\end{matrix}\right)\end{matrix}\right]$$

Yes, the formula becomes quite complicated, and if we have multiple cache systems it becomes even more difficult to comprehend - hence the complication of comprehending reliability of systems with cache.

Retry Pattern

The retry pattern is a simple way to handle some failures and improve reliability. Why some? As a general rule, retry handles momentary outages we denote here as risk of load and latency, but fail to handle (in most cases) the outages due to risk of change.

Outages caused by load and latency tend to be momentary - and another try even 1ms later can succeed where the previous one failed. However, outages caused by change tend to take minutes to tens of minutes to fix, and in some cases more. As a result, retrying a few milliseconds later will most probably hit the same outrage caused by change.

Retry also implies the operation to retry is idempotent - it is designed to be retied. Read operations are idempotent by definition, while operations with side effects, persistence, data updates, inserts or deletes are by default not idempotent. However, in some cases such operations can be modelled as idempotent.

Mutation operations can be made idempotent using an operation key by checking if an operation with the key already happened. While this sounds very generic, we all know a few implementations of this general idea - such as using a transaction id, optimistic locking or update by insert with client provided primary key value. The common among all is that if we retry, while the original request has succeeded, the second can be detected using the key

value and handled accordingly (for instance, prevent the double doing while returning a success to the client).

- **Transaction_id** - the providing service on receiving a mutation operation with a transaction id, first writes it to a write ahead log and then processes the mutation to other tables. If the transaction id already exists in the write ahead log, a duplicate key error is potentially ignored.
- **Optimistic locking** - the providing service checks the client supplied version, and if that version does not match the current version + 1, it rejects the operation. On retry, if the previous attempt worked, the version number has already advanced and the second try will fail.
- **Update by insert** - The providing service does an insert into a log table (similar to transaction id) instead of an update. For instance, consider an inventory system that on checkout updates the inventory to have one less product. An update operation of `update inventory set product_inventory = product_inventory - count` is not idempotent. However, an insert operation of `insert into ordered_inventory (product_id, order_id, count) values (?, ?, ?)` is idempotent if we have a primary key or unique index on `product_id, order_id`.

What is the effect of retry on predicted reliability? First, it only affects the 'risk of load and latency' component, and then it reduces the risk divided by the number of retries.

$$\left(\begin{array}{c}\text{Risk of Load}\\ \text{and Latency}\\ \text{with Retry}\end{array}\right) = 1 - \frac{1 - \left(\begin{array}{c}\text{Risk of Load}\\ \text{and Latency}\\ \text{without Retry}\end{array}\right)}{\left(\begin{array}{c}\text{Number of}\\ \text{Retries}\end{array}\right)}$$

Fallback Pattern

The fallback pattern recognizes that risk exists and things can fail. Once something does fail, the system is designed to do something different, such as fallback to another service, to another product flow or to shutdown a feature.

Fallback can be implemented as

- Fallback of a certain microservice can be an equivalent microservice deployed into another data center, another cloud, another zone (handles risk of load and latency)
- Fallback of a certain microservice can be a delayed deployed microservice, such as 1 week ago version (handles risk of change, yet requires forward / backward compatibility)
- Fallback of a certain microservice can be another implementation of the microservice using another software stack or another SaaS provider, like another payment provider, another type of cloud database, another email provider, etc. We can have multiple providers with different transaction prices, trying the more cost effective first, and if that fails, use the more costly option.
- Fallback can be by changing functionality - if we detect the email service fails, we can try the SMS service instead.
- Fallback can be by shutting down a feature - if the coupon service fails, we can block the ability to add coupons on checkout. If the shipping calculator service fails, we can show the cart without shipping cost and show a message that shipping costs will be provided using email soon, or we can use a generic average shipping code.

The impact of a fallback pattern improves reliability by handling the error cases, and turning them into another valid flow. While the fallback itself can fail, the reality is that the end result is way more reliable.

The predicted reliability of a system with a fallback becomes

$$\left(\begin{array}{c}\text{Predicted}\\ \text{reliability}\end{array}\right) = \left(\begin{array}{c}\text{Predicted Reliability}\\ \text{of system upstream}\\ \text{of the fallback}\end{array}\right) \times \left[\left(\begin{array}{c}\text{Predicted Reliability of}\\ \text{system downstream of}\\ \text{the fallback}\end{array}\right) + \left(1 - \left(\begin{array}{c}\text{Predicted Reliability of}\\ \text{system downstream of}\\ \text{the fallback}\end{array}\right)\right) \times \left(\begin{array}{c}\text{Fallback System}\\ \text{Predicted Reliability}\end{array}\right)\right]$$

In the formula above, we break the system to anything before the microservice with a fallback pattern implemented (upstream system) and anything after (downstream system). The fallback only handles features of the downstream system, compensating given its own reliability.

Circuit Breaker

The Circuit Breaker pattern is a pattern to ensure a downstream subsystem load does not exceed a certain limit, or that a specific customer / tenant / client consumes all the resources and creates a “denial of service” equivalent on the downstream system.

When a circuit breaker is activated, it is knowingly causing errors, like 429 too many requests or 504 timeouts. Circuit Breaker creates a clear cutoff point at which client services experience and can expect a failure, falling back to another flow.

Consider for instance a personalized recommended products service, which gives product recommendations adapted specifically for each user on a product page. If this service fails or is overloaded, a circuit breaker can be activated to prevent additional load on the service. A fallback can be to hide the recommended products section in the product page, or show a cached generic (not personalized) recommended products.

We have to ask - why is Circuit Breaker considered a reliability pattern if it creates errors? The Circuit Breaker protects a downstream system from resource exhaustion and improves the reliability for other clients of the downstream system.

Circuit Breaker can be implemented as

- A limit of API calls per client per period of time.
- A backoff on the number of errors or rate of errors from the downstream system, triggering the circuit breaker for a period of time.
- A backoff on downstream system load, such as downstream system returns 429 or 504 HTTP status (or equivalent), enabling the downstream system cooldown for a period of time.

Summary

"When you have a monolith, you've got one big problem to solve. Switch to microservices, and now you've got 99 smaller problems—plus a distributed system". Distributed systems are hard. Two challenges with distributed systems are data consistency and reliability.

This playbook reviews microservices reliability, including how to predict and how to measure and alert on microservices reliability, and a list of architectural patterns to regain reliability given a distributed microservices system.

Beyond the Playbook

Thank you for diving into the Microservices Reliability Playbook. Building reliable systems is a journey, and I hope these strategies spark new ideas for your teams. Let's keep the conversation going—share your thoughts, questions, or experiences with microservices reliability! Reach out to me on X (@YoavAbrahami) or LinkedIn (linkedin.com/in/yoavabrahami) using #MicroservicesPlaybook. Whether it's a tough challenge you've faced or a win you've celebrated, I'd love to hear from you and discuss how we can push the boundaries of resilient engineering together."

About the Author



Yoav Abrahami is Wix Enterprise CTO, where he drives innovation for a platform serving over 250 million users. With more than 30 years in technology, including his tenure as Wix's first Chief Architect, Yoav has shaped the company's scalable, cloud-native infrastructure, introducing CI/CD and TDD practices that empower engineering teams. Yoav authored the Microservices Reliability Playbook to share practical strategies for resilient systems, distilled from decades of solving real-world scalability challenges. Based in Tel Aviv, he enjoys hiking and mountain biking when not advancing developer-centric solutions.

https://www.linkedin.com/in/yoavabrahami/

https://x.com/yoavabrahami