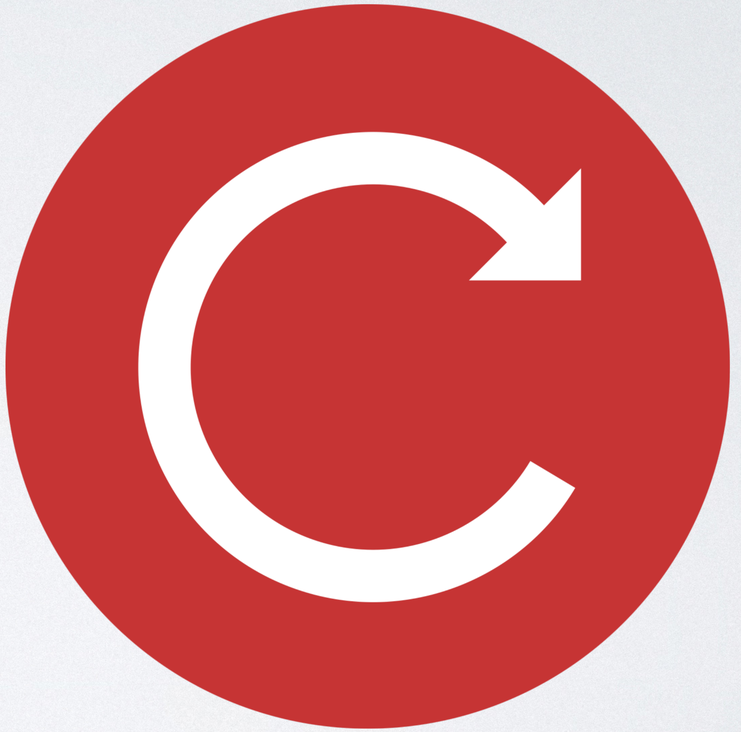

# RETRY PATTERN 🔁

by Enrico Pezzano

# INTRODUCTION 🎬

- “Resilient approach”
- Robustness & tolerance is a paramount
- Powerful & widely adopted
- Systematic & flexible
  - ⚡
    - Reliability & availability

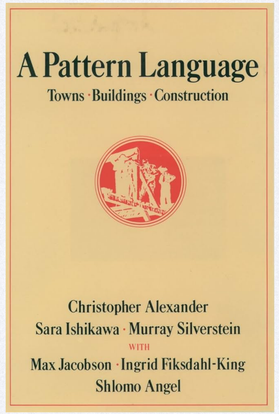


# OUTLOOK 😳

- Why do we need it ?
- What is it ?
- What is the problem?
- Pros & Cons ?
- Solutions …
- Code examples

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

–Christopher Alexander, A Pattern Language 📖



# WHAT IS IT ❓🤨

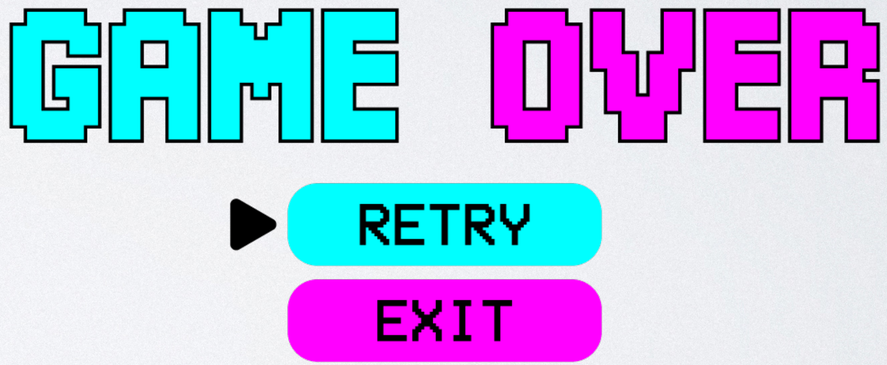
⚡ Specific strategy

⏰ Issues, external dependencies, etc

📲 Retry logic incorporated into the app

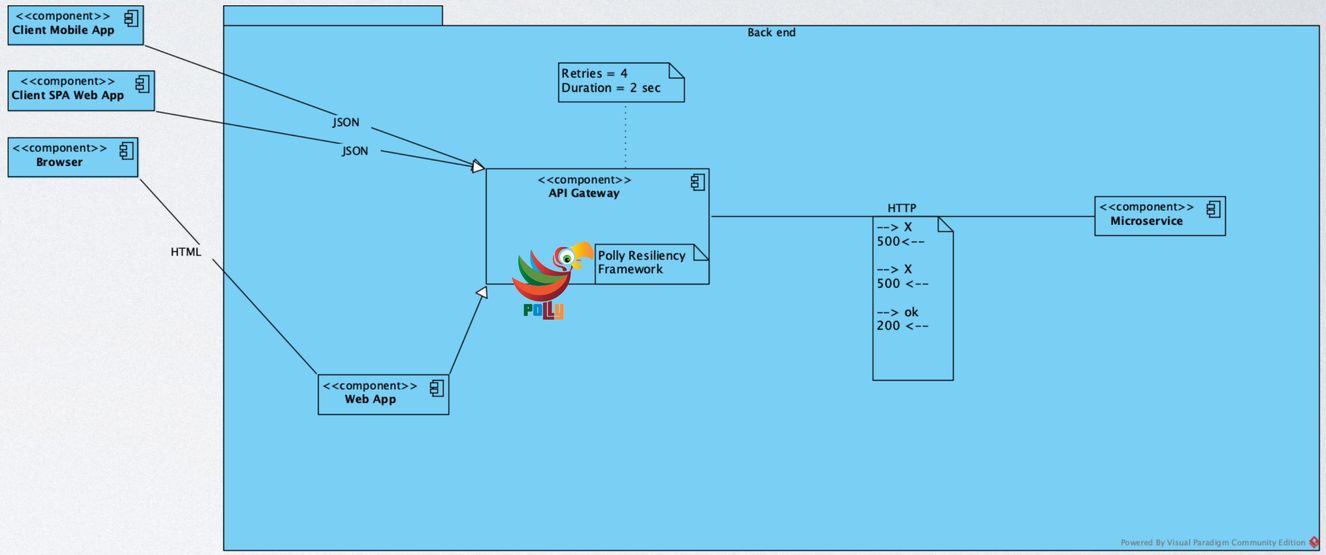
💔 Enhance system recovery from failures

🐆 Without jeopardising functionality



# A DESIGN PATTERN FOR μ-SERVICES

- Common pattern used also in a lot of others environments (e.g. …)
- # of times it should retry, delay between attempts and actions to take
- Typically self-correcting → if repeated, it's likely to success 🔑💡

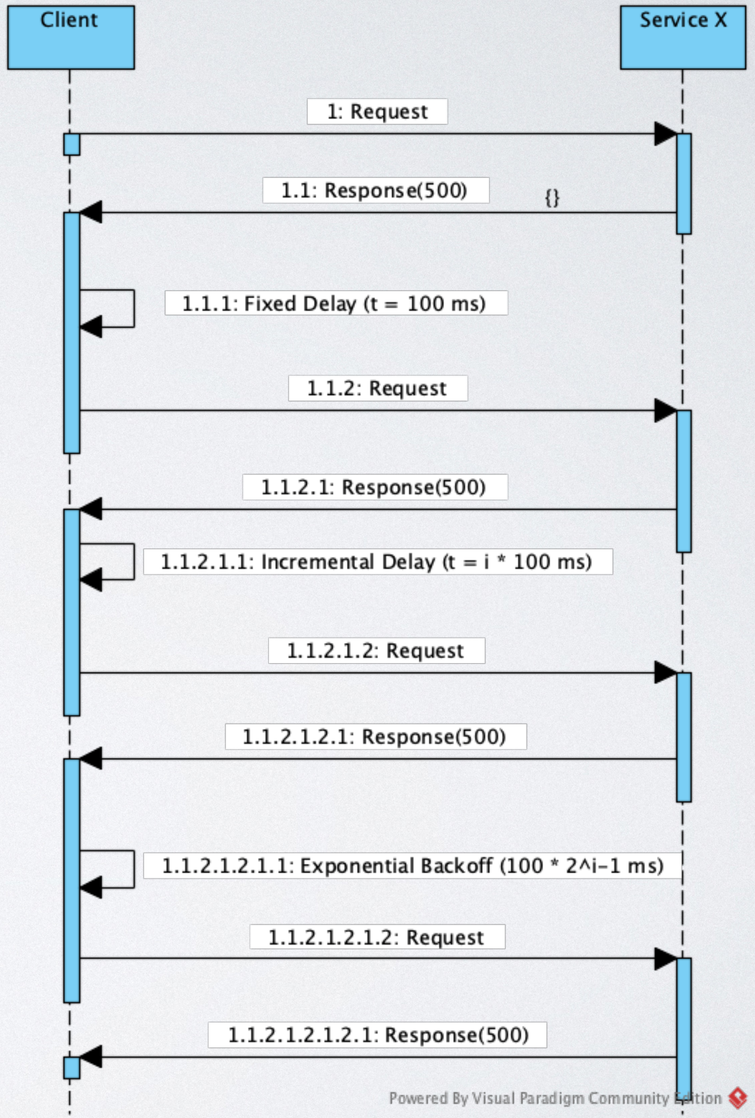



**Keep in mind that an application should be designed to handle these types of situations elegantly and transparently**

# STRATEGIES 

- Retry logic:
    1. Retry
    2. Retry after delay
- Exponential backoff
- Circuit breaker
- Cancel

**POV:**
**if overwhelming → longer to recover**



# ISSUES AND CONSIDERATIONS 🤔

❌ Without a threshold, a bigger degrade is possible

❌ Responsiveness is at risk, if continually trying to perform a failing operation

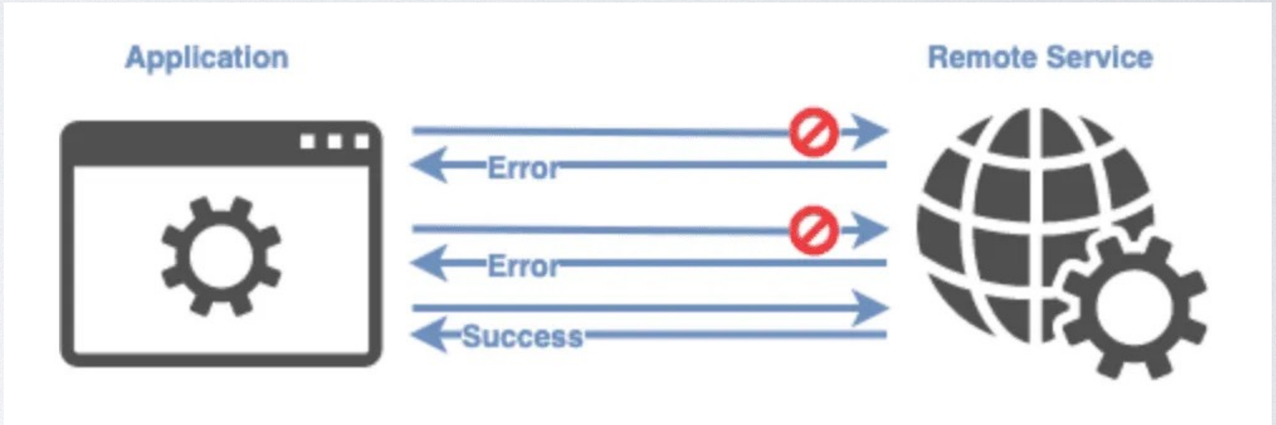
⭕ Sometimes it's better to fail fast

✅ If exhausted resources → better solution to scale out

✅ If operation is idempotent, it's safe to retry. Otherwise…

⭐ Enhanced reliability & user experience

⭐ Adaptability to dynamic environments




**An idempotent system ensures that the actual execution on this request only happens once (doesn't matter the requests number)**

# “ONE SHOE FITS ALL” SUITABILITY? 👞🎯

✅ Transients faults

✅ Short lived and repeating a (failed) request

❌ Long lasting faults → retrying damage the responsiveness

❌ Non-transient faults …

❌ Busy faults frequently → better to scale out

…if all you have is a hammer, everything looks like a nail…

- Failures due to various reasons and various natures

    1. Some exceptions can be resolved quickly

    2. Some are longer lasting

👉Retry policy to adjust everything

# FAQ 📣

- How do you handle idempotence when using the retry pattern?
- What are common challenges when implementing the retry pattern?
- What monitoring and logging practices are recommended for the retry pattern?
- Are there best practices for implementing the retry pattern in **specific programming languages** or **frameworks**?

```
private int retryCount = 3;
private readonly TimeSpan delay = TimeSpan.FromSeconds(5);

public async Task OperationWithBasicRetryAsync()
{
  int currentRetry = 0;

  for (;;)
  {
    try
    {
      // Call external service.
      await TransientOperationAsync();

      // Return or break.
      break;
    }
    catch (Exception ex)
    {
      Trace.TraceError("Operation Exception");

      currentRetry++;

      // Check if the exception thrown was a transient exception
      // based on the logic in the error detection strategy.
      // Determine whether to retry the operation, as well as how
      // long to wait, based on the retry strategy.
      if (currentRetry > this.retryCount || !IsTransient(ex))
      {
        // If this isn't a transient error or we shouldn't retry,
        // rethrow the exception.
        throw;
      }
    }

    // Wait to retry the operation.
    // Consider calculating an exponential delay here and
    // using a strategy best suited for the operation and fault.
    await Task.Delay(delay);
  }
}

// Async method that wraps a call to a remote service (details not shown).
private async Task TransientOperationAsync()
{
  ...
}
```

```
private bool IsTransient(Exception ex)
{
  // Determine if the exception is transient.
  // In some cases this is as simple as checking the exception type, in other
  // cases it might be necessary to inspect other properties of the exception.
  if (ex is OperationTransientException)
    return true;

  var webException = ex as WebException;
  if (webException != null)
  {
    // If the web exception contains one of the following status values
    // it might be transient.
    return new[] {WebExceptionStatus.ConnectionClosed,
                  WebExceptionStatus.Timeout,
                  WebExceptionStatus.RequestCanceled }.
            Contains(webException.Status);
  }

  // Additional exception checking logic goes here.
  return false;
}
```

We can model a recoverable scenario by instantiating `FindCustomer` like this:

```java
final var op = new FindCustomer(
    "12345",
    new CustomerNotFoundException("not found"),
    new CustomerNotFoundException("still not found"),
    new CustomerNotFoundException("don't give up yet!")
);
```

In this configuration, `FindCustomer` will throw `CustomerNotFoundException` three times, after which it will consistently return the customer's ID ( `12345` ).

In our hypothetical scenario, our analysts indicate that this operation typically fails 2-4 times for a given input during peak hours, and that each worker thread in the database subsystem typically needs 50ms to "recover from an error". Applying these policies would yield something like this:

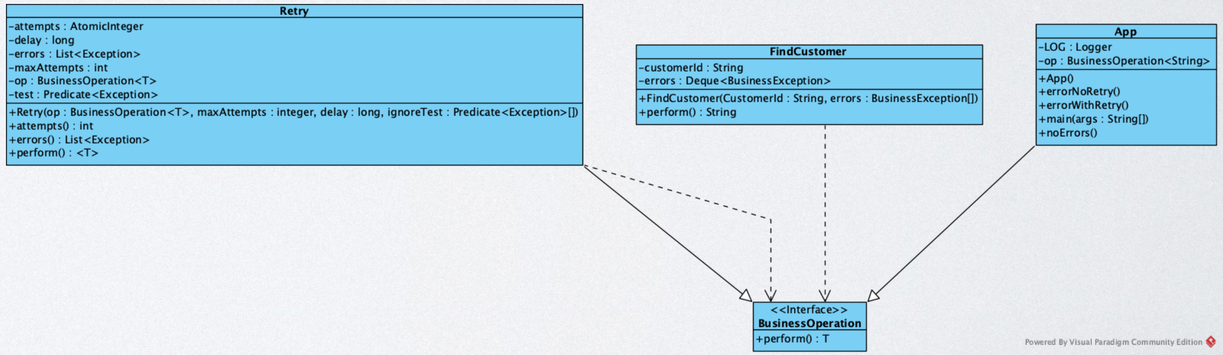
```java
final var op = new Retry<>(
    new FindCustomer(
        "1235",
        new CustomerNotFoundException("not found"),
        new CustomerNotFoundException("still not found"),
        new CustomerNotFoundException("don't give up yet!")
    ),
    5,
    100,
    e -> CustomerNotFoundException.class.isAssignableFrom(e.getClass())
);
```

In our hypothetical application, we have a generic interface for all operations on remote interfaces.

```java
public interface BusinessOperation<T> {
  T perform() throws BusinessException;
}
```

And we have an implementation of this interface that finds our customers by looking up a database.

```java
public final class FindCustomer implements BusinessOperation<String> {
  @Override
  public String perform() throws BusinessException {
    ...
  }
}
```



```
class RetryTest {

    private var success = "success"

    @Mock
    private lateinit var externalService: ExternalService

    private val logger: LoggerWrapper = LoggerWrapper()
    private lateinit var retry: Retry<String>

    @Before
    fun setUp() {
        MockitoAnnotations.initMocks(this)
    }

    @Test
    fun should_not_retry_when_successful() {
        retry = Retry(logger)

        val result = retry.run { success }

        assertThat(result).isEqualTo(success)
        assertThat(retry.retryCounter.get()).isEqualTo(0)
    }

    @Test
    fun should_retry_once_then_succeed_when_first_fails_and_second_successes() {
        retry = Retry(logger)
        `when`(externalService.run())
            .thenThrow(RuntimeException("Something was wrong..."))
            .thenReturn(success)

        val result = retry.run(externalService::run)

        assertThat(result).isEqualTo(success)
        assertThat(retry.retryCounter.get()).isEqualTo(1)
    }

    @Test
    fun should_throw_exception_when_max_retries() {
        retry = Retry(logger)
        `when`(externalService.run())
            .thenThrow(RuntimeException("Something was wrong..."))
            .thenThrow(RuntimeException("Something was wrong..."))
            .thenThrow(RuntimeException("Something was wrong..."))

        assertThatThrownBy {
            retry.run(externalService::run)
        }.isInstanceOf(RuntimeException::class.java)
            .hasMessage("Command fails on all of 3 retries")

    }
}
```

```
   @Configuration
@EnableRetry
class Application {
    @Bean
    fun service(): Service {
        return Service()
    }
}
```

```
  @Service
  class Service {
    @Retryable(maxAttempts = 2, include = [RemoteAccessException::class])
    fun service() {
           // ... do something
           logger.info("Success calling external service")
       }
       @Recover
       fun recover(e: RemoteAccessException) {
           // ... do something when call to service fails
           logger.error("Recover for external service")
       }
   }
```

```
INFO 81668 --- [ Logge : RetryService ] - Calling external service...
INFO 81668 --- [ Logge : RetryService ] - Success calling external service
INFO 81668 --- [ Logge : RetryService ] - Calling external service...
INFO 81668 --- [ Logge : RetryService ] - Success calling external service
INFO 81668 --- [ Logge : RetryService ] - Calling external service...
INFO 81668 --- [ Logge : RetryService ] - Calling external service...
ERROR 81668 --- [Logger : RetryService ] - Recover output
```

# BEST PRACTICES 🏆

⚠️ Log all connectivity failures

⚠️ Ensure that all retry code is fully tested against a variety of failure conditions

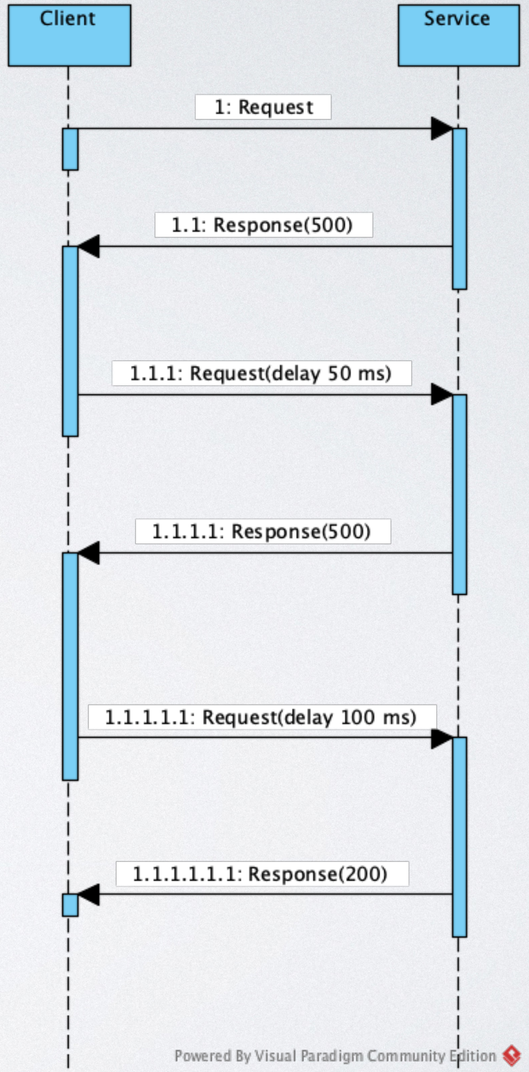
⚠️ Investigate the faults that are most likely to occur

⚠️ Avoid amplifying retries at multiple levels

⚠️ Consider a server-wide retry budget

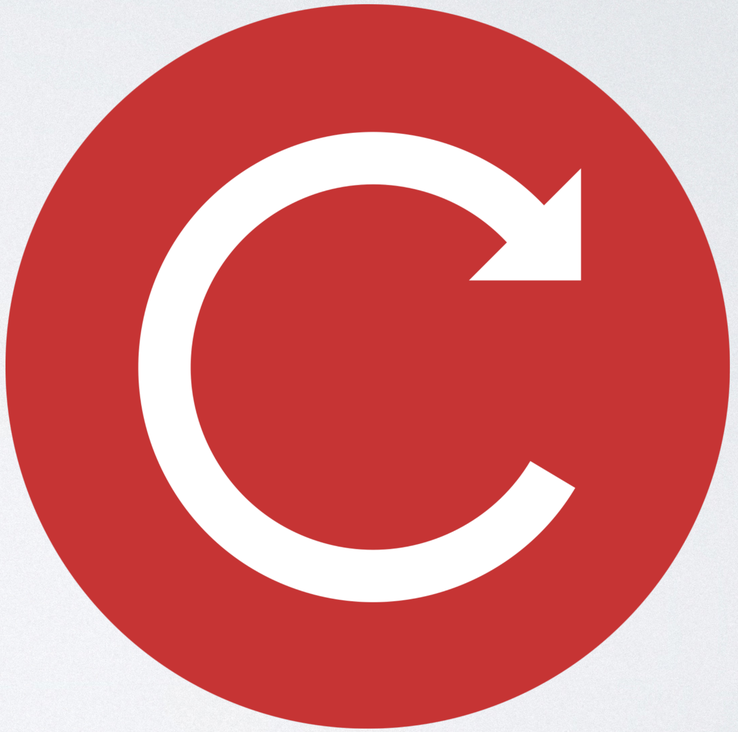
⚠️ Ensure retries aren't done on operations where consumers are not waiting for response

⚠️ Never wait synchronously between retry attempts. For example, use "*Task.Delay*" in place of "*Thread.Sleep*" in .Net



# CONCLUSIONS 


- Transient faults with a remote service or resources → do take care of what the error code is before retries
- We don't want to be victim of retry anti-patterns
- Incorporating intelligent retry logic, exponential backoff and circuit breaker mechanism, developers can create robust systems capable of recovering from unexpected hiccups
- As technology continues to evolve, the RP remains a valuable tool of developers



# REFERENCES ※📚


- Online post:
  - Author: Vijaya Lakshmi, 2022
  - Title: Design patterns for micro-services
  - Website: https://www.linkedin.com/pulse/vijaya-lakshmi-v/
- Medium article:
  - Author: Harish Bhattbhatt, 2021
  - Title: Best practices for retry pattern
  - Website: https://harish-bhattbhatt.medium.com/best-practices-for-retry-pattern-f29d47cd5117
- Microsoft Article:
  - Year: 2023
  - Title: Transient fault handling
  - Website: https://learn.microsoft.com/en-us/azure/architecture/best-practices/transient-faults
- Google Article:
  - Author: Mike Ulrich, 2017
  - Title: Addressing Cascading Failures
  - Website: https://sre.google/sre-book/addressing-cascading-failures/
- Microsoft Documentation:
  - Title: Retry pattern
  - Website: https://learn.microsoft.com/en-us/azure/architecture/patterns/retry
- Journal article:
  - Author: Kamini Kamal, 2023
  - Title: Understanding the Retry Pattern
  - Journal: Medium
  - Website: https://blog.bitsrc.io/retry-pattern-3f7221913e41?gi=161787a102da
- AWS Article:
  - Author: Marc Brooker, 2015
  - Title: Addressing Cascading Failures
  - Website: https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

🥰

? 👊 ?