Building resilient applications on Azure

Resilience patterns and validation – a sea of possibilities

Jan Pollack & Ricardo Niepel

Technical Specialists for Azure App Innovation

AGENDA

01 How can cloud native applications fail?

02 How can you build your application resilient to failure?

03 How can I implement and validate a resilient application architecture?

“Anything that can go wrong will go wrong”

Hopefully not!

„... *and at the worst possible time*” - Edward A. Murphy Jr, Murphy's law from ~1948


76-035
Ohlmeier

“

1. The network is reliable
2. Latency is zero

... ”

How can cloud native applications fail?

L. Peter Deutsch, 8 Fallacies of distributed computing from 1994

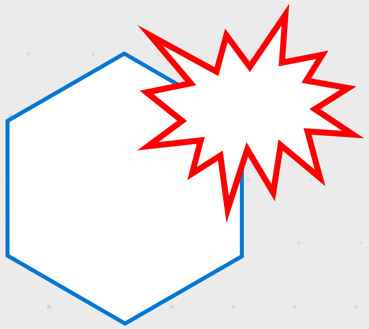
Reliable workloads are:

Highly available (HA) – amount of uptime a system is ready to perform core or essential functions

Resilient - able to recover gracefully from failures and continue to function with minimal downtime and data loss

VIEWS ON FAILURE

Failure of a component

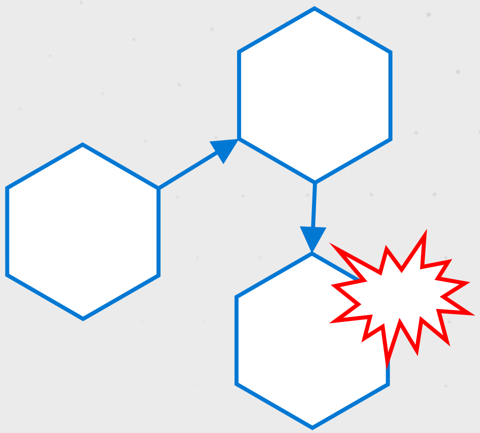


Focus on: **Minimizing risk of failure**

Questions to ask:

- How to implement error handling?
- What service SKU to select?
- What redundancy level to choose?

Failure of a dependency

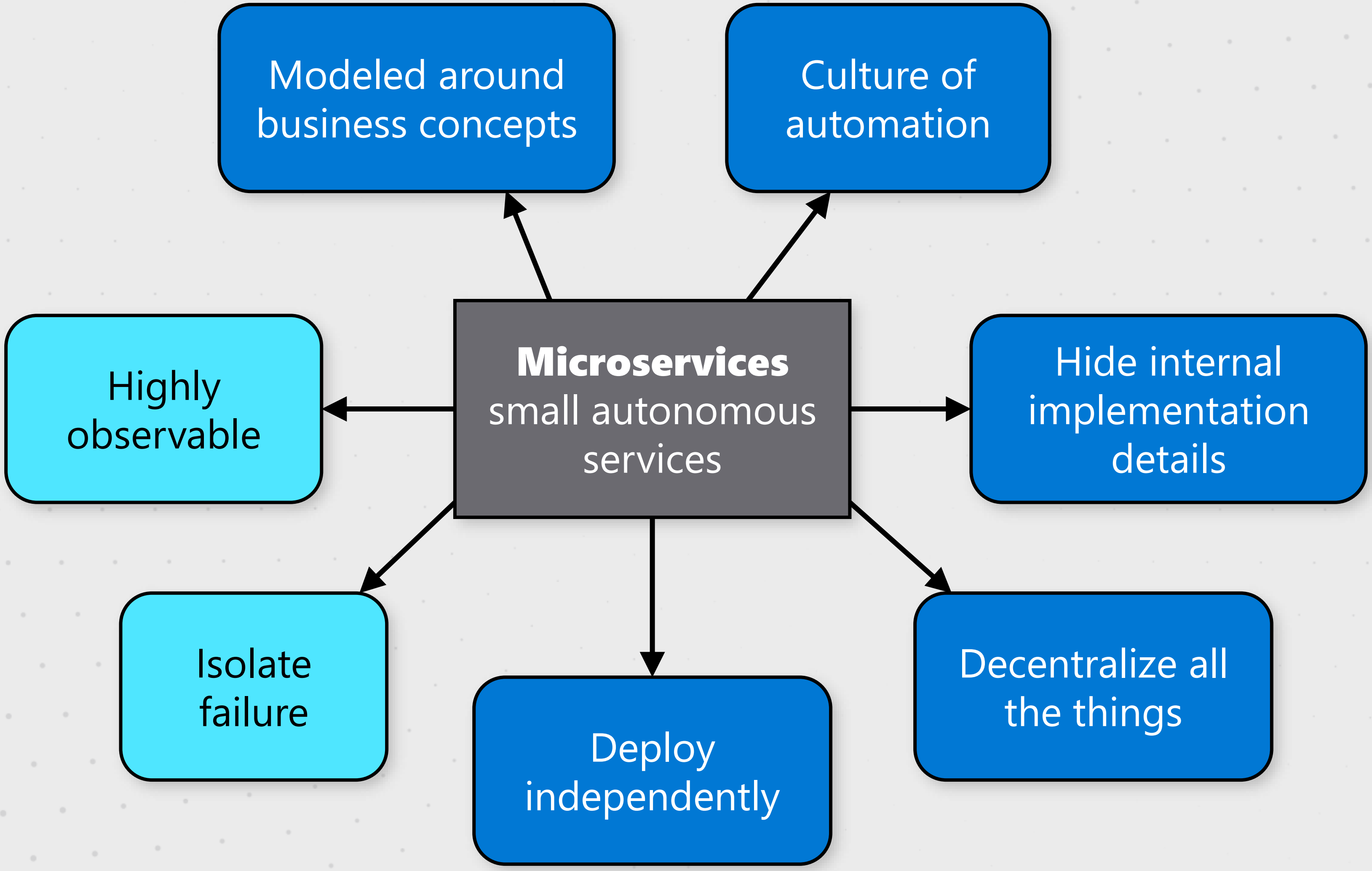


Focus on: **Detecting and isolating failures**

Questions to ask:

- How to detect failures?
- How to protect myself?
- How to support recovery?

PRINCIPLES OF MICROSERVICES



Sam Newman, Building Microservices from 2014

INFRASTRUCTURE FAILURES ARE RARE BUT CAN HAPPEN

Zonal failure

Some compute will be completely unavailable

Load Balancer failure

Load balancing from the outside to some nodes will fail

Node failure

A compute node will fail completely

Disk failure

What is the max duration of data loss that is acceptable?

>

Can be solved by having sufficient available distributed infrastructure resources

<

PARTIAL INFRASTRUCTURE FAILURES ARE MORE COMMON

Health probe unreliability

Service or health probe do not reflect true state

Node overload

Service runs but cannot respond in a timely manner due to CPU/ memory/ disk

Port exhaustion

Service works but cannot connect to its dependencies

DNS resolution failures

Service works but cannot resolve its dependencies

>

Needs to be solved by application design to detect and handle accordingly

<

Handle transient failures

When failure is encountered, retry, queue and orchestrate transactions/operations

Continue operations during failure

Some part of the system will always be in a failed state.

Maintain consistent state

Apply compensating logic across multiple data stores or attempt resubmission until operation succeeds.

Bing Image Creator: Little ship in a bad storm in the middle of the sea

Demo

Contonance - Awesome Ship Maintenance

"If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail."

How can you build your application resilient to failure?

Abraham Maslow, Law of the instrument from 1966

RETRY

MAYBE IT'S JUST A BLIP

"Transient errors?!"

Solution

Transparently handle failures as if the failure condition will resolve itself.

Considerations

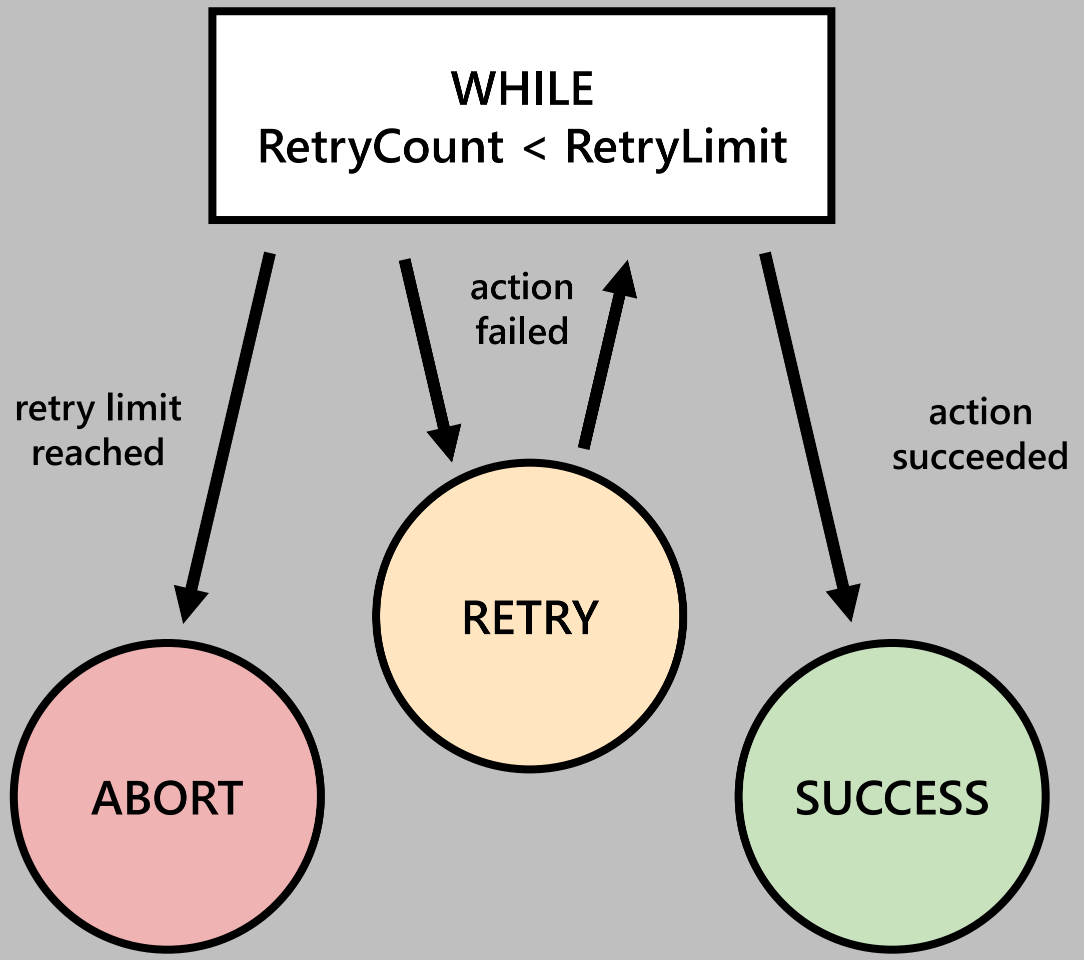
Only works if you can identify transient errors.

Use exponential back-off to allow for retries to be made initially quickly, but then at progressively longer intervals, to avoid hitting a subsystem with repeated frequent calls if the subsystem may be struggling.

Apply jitter to prevent retries bunching into further spikes.

Beware

Only retry idempotent operations or provide compensating logic.



TIMEOUT

DON'T WAIT FOREVER

"DNS failure?!"

Solution

Only allow an operation a limited amount of time to complete.

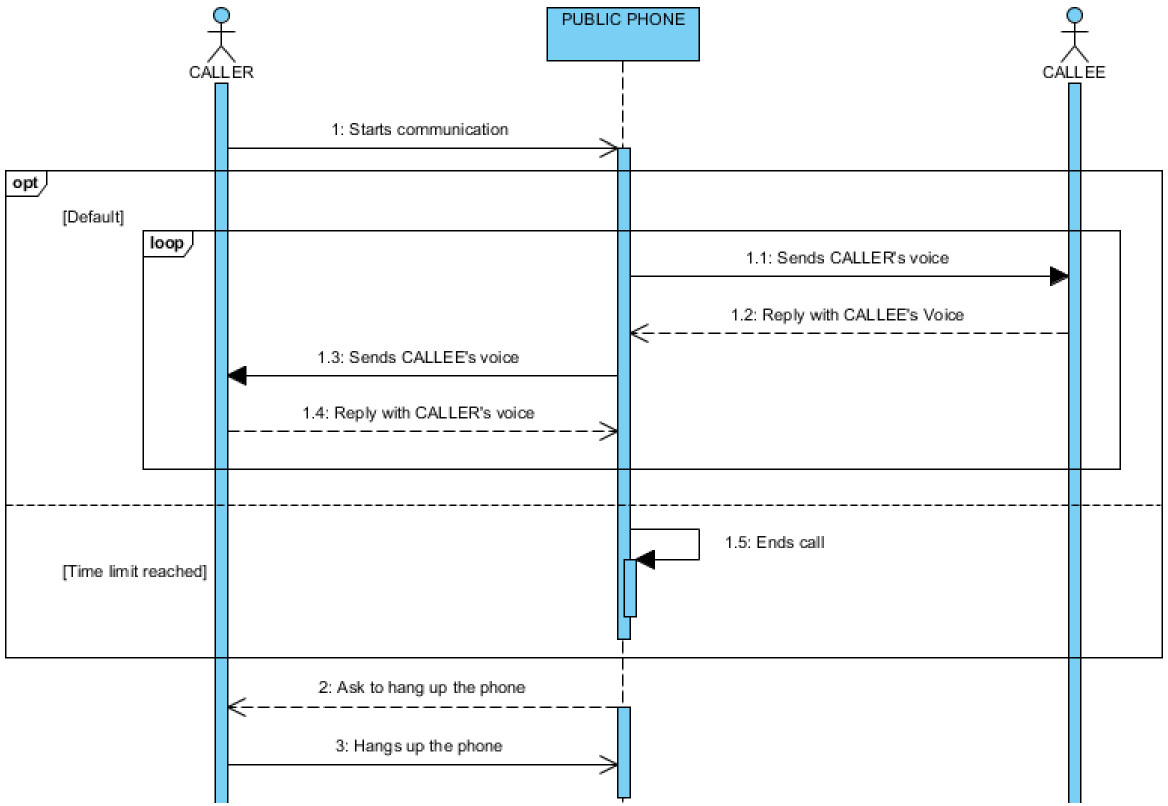
Waiting forever (having no timeout) is a bad design strategy. It leads to the blocking up of threads or connections (itself often a cause of further failure), during a faulting scenario.

Considerations

May require operation to run on an additional thread, if it's not cancellable, doubling the number of threads you actually require.

Beware

Easy with cancellable operations. If this isn't possible, Timeouts can become quite tricky.



CIRCUIT BREAKER

STOP HITTING IT IF IT HURTS

"Dependency failure?!"

Solution

Prevent an application from repeatedly trying to execute an operation that is likely to fail and enables an application to detect whether the fault has been resolved.

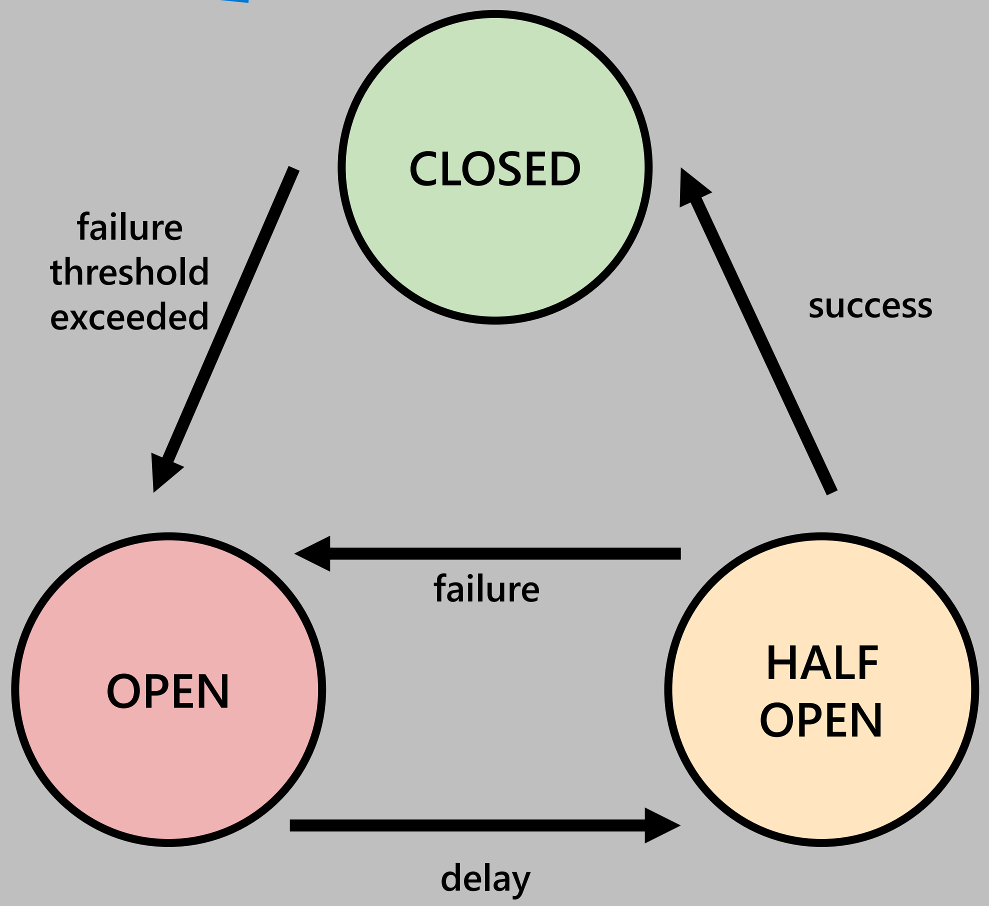
If the problem appears to have been fixed, the application can try to invoke the operation.

Considerations

Circuit Breaker isn't just a more elaborate Retry—it's based on the premises that subsequent calls will likely *not* succeed, whereas Retry assume they will likely succeed.

Beware

Circuit Breaker are inherently stateful.



THROTTLING / RATE LIMITING

ENOUGH IS ENOUGH

“Heavy load?!”

Solution

Allow applications to use resources only up to a limit, and then throttle them when this limit is reached.

This can allow the system to continue to function and meet service level agreements, even when an increase in demand places an extreme load on resources.

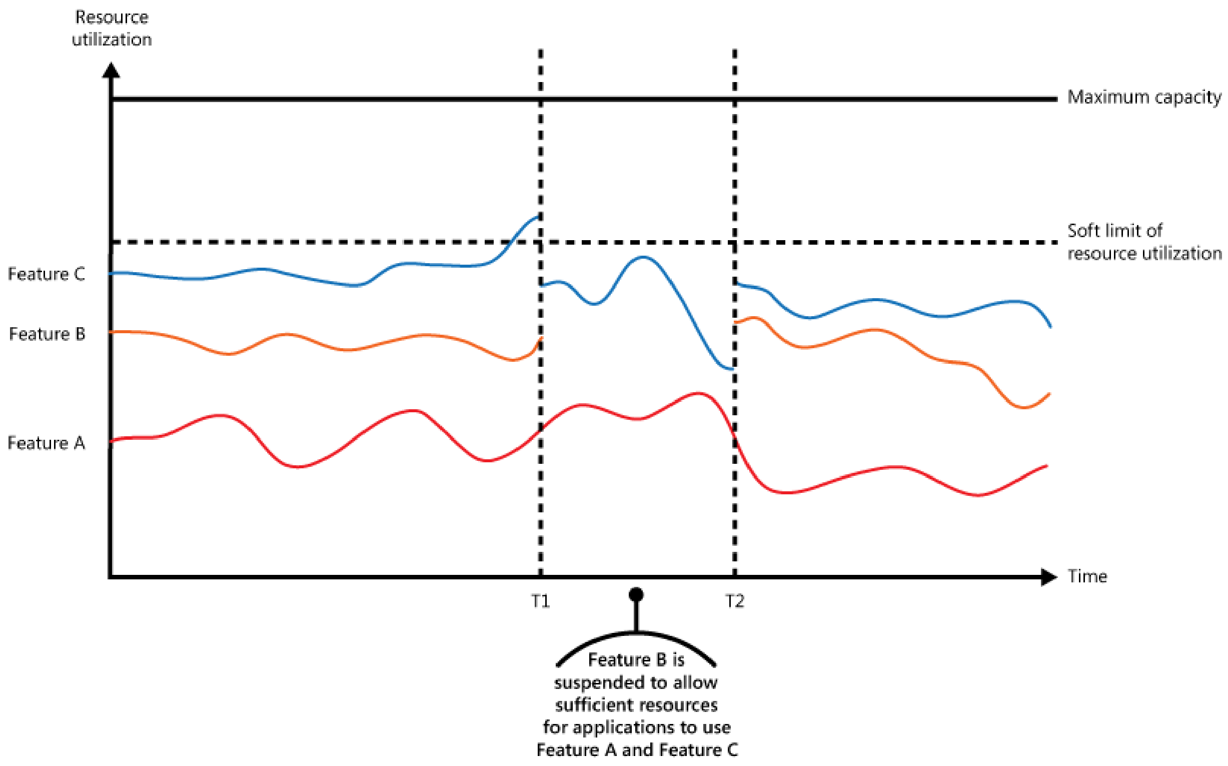
Considerations

Often used as a building block for a higher-level larger pattern (e.g. bulkhead).

Rate limiting (i.e. calls/sec) or leasing (i.e. a pool of *n* resources) are typical stand-alone versions of Throttling.

Considerations

Familiarize yourself with the Throttling behaviors of Azure services and APIs!



Bulkheads of the vehicle transport ship USNS GILLILAND, U.S. National Archives

BULKHEAD

MAYBE IT'S JUST A BLIP

"Greedy parts?!"

Solution

Limits the amount of resources different parts of your application can consume.

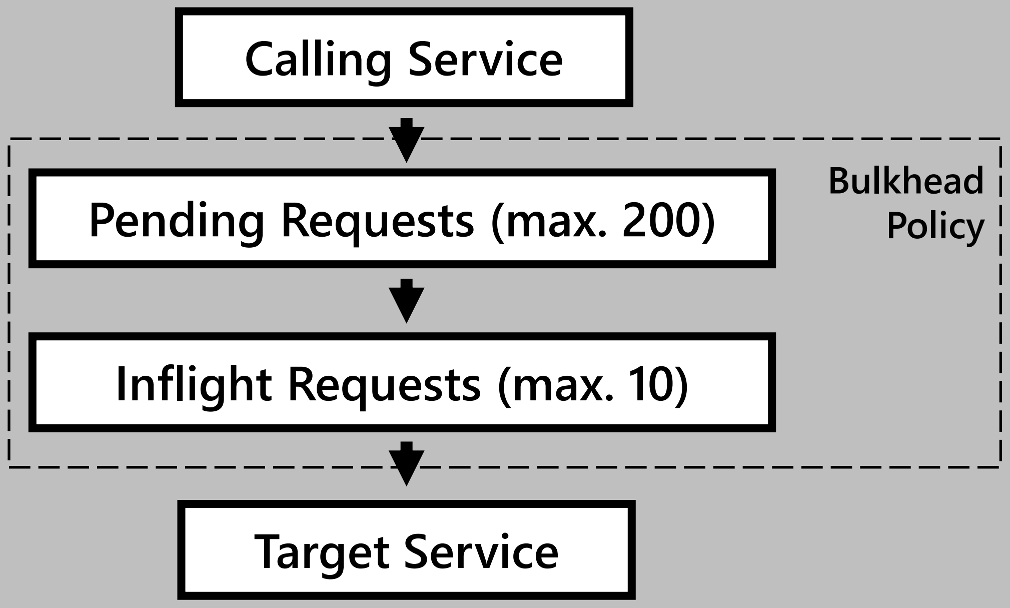
Bulkheads prevent single unresponsive components or instances from bringing down the whole system.

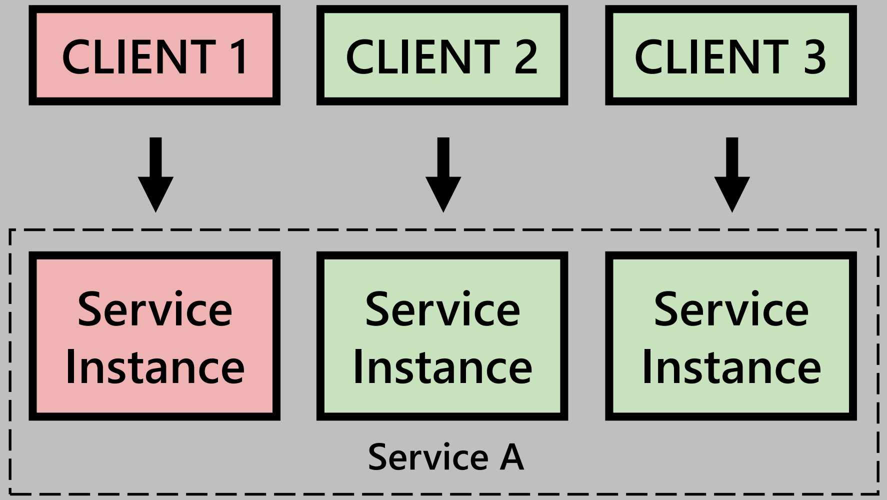
Considerations

Bulkhead can be implemented in various ways, depending on the requirements and solution.

Beware

Finding the right grouping granularity can be difficult.





QUEUE-BASED LOAD LEVELING

KEEP CALM AND DISPATCH A MESSAGE

"Different processing speed?!"

Solution

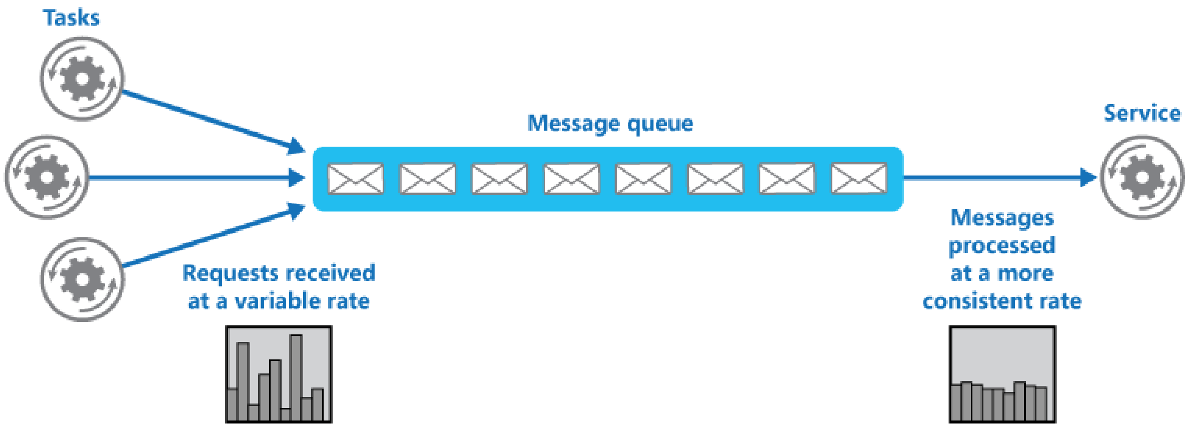
Introduce a queue between the task and the service.
The task and the service run asynchronously
at their own processing speed.

Considerations

Requires a separate channel for clients
to receive a task's result (i.e. pub/sub, web sockets, etc).

Beware

Introduces a new dependency (i.e. the queue)
that may fail as well.



FALLBACK / GRACEFUL DEGRADATION

IT'S BETTER TO DO SOMETHING THAN TO DO NOTHING

"Alternatives?!"

Solution

Establishes an action plan during a failure rather than leave it to have unpredictable effects on your system.

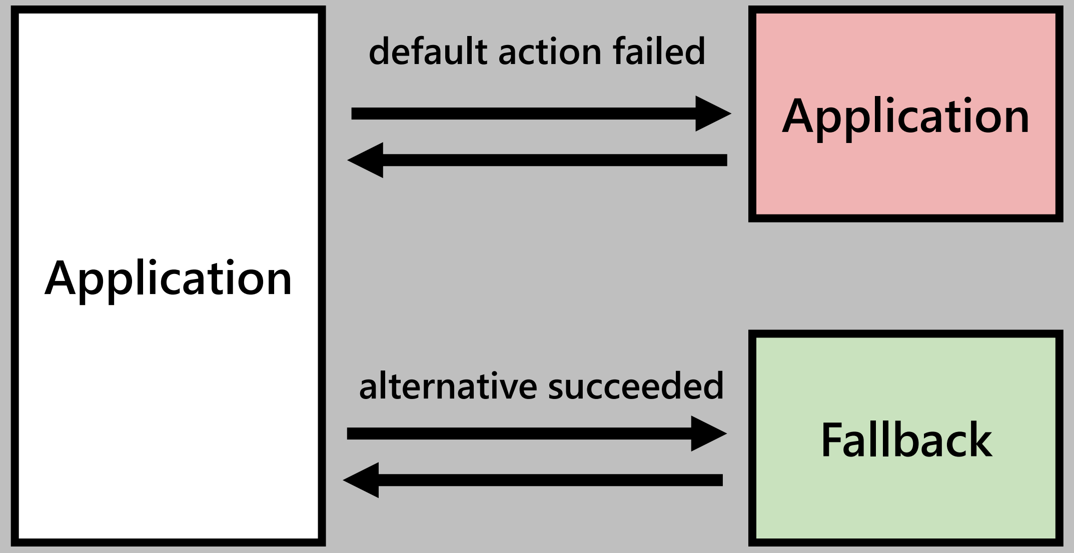
Considerations

Defines an alternative value to be returned or action to be executed in case of a failure.

Does not need to be complicated, sometimes a good user experience is just enough.

Beware

Could add significant costs and latency to the system.



HEDGING

WE TRY EVERYTHING WE CAN

"Latency sensitive?!"

Solution

defines multiple backup solutions when an operation fails.

Considerations

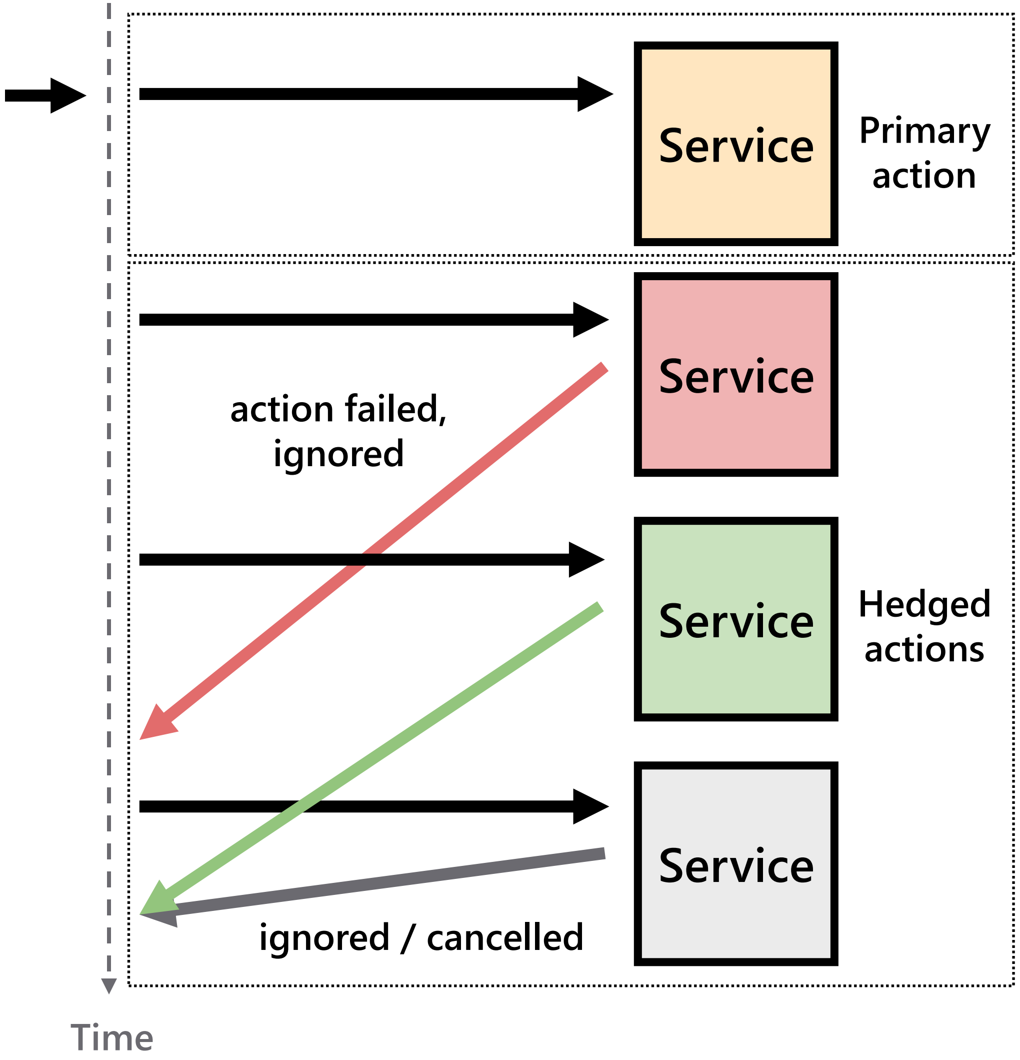
Hedging means spawning multiple concurrent execution paths to substitute a possible failure on the primary path.

Initiates a primary action, waits a predefined interval.
After the interval, a parallel request as backup is made.
It waits for the first successful request to complete and return it. Any other hedged ongoing request will terminate.

A common case is the use of multiple endpoints of a service for retrieving a resource.

Beware

It has as primary goal to improve latency,
but it is load expensive.



“Everything fails, all the time”

How can I implement and validate a resilient application architecture?

Werner Vogels, Chief Technology Officer of Amazon, from 2008

Demo

Implementation of **Resilience Patterns**

RESILIENCY WHEN USING GENERATIVE ARTIFICIAL INTELLIGENCE

Azure OpenAI, GPT, ChatGPT, LLaMA …

Reasons to fail…

- Transient HTTP errors (still applies)
- Service failures (still applies)
- Hit rate limits
- Throttling
- Price sensitivity
- Context input limits

```
// Semantic Kernel - Retry Mechanisms
var semanticKernel = Kernel.Builder.WithRetryBasic(
    new BasicRetryConfig
    {
        MaxRetryCount = 3,
        UseExponentialBackoff = true,
        MinRetryDelay = TimeSpan.FromSeconds(2),
        MaxRetryDelay = TimeSpan.FromSeconds(8),
        MaxTotalRetryTime = TimeSpan.FromSeconds(30),
        RetryableStatusCodes = new[] { HttpStatusCode.TooManyRequests,
                                       HttpStatusCode.RequestTimeout },
        RetryableExceptions = new[] { typeof(HttpRequestException) }
    })
    .Build();


# LangChain - Fallback Mechanisms
openai_llm = ChatOpenAI(max_retries=0)
anthropic_llm = ChatAnthropic()
fallback_anthropic = openai_llm.with_fallbacks([anthropic_llm])

short_llm = ChatOpenAI()
long_llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
fallback_long = short_llm.with_fallbacks([long_llm])

fallback_nonchat = chatopenai_chain.with_fallbacks([openai_chain])
fallback_nonchat.invoke({"animal": "turtle"})

fallback_4 = chat_prompt | openai_35.with_fallbacks([openai_4])
```

CHAOS ENGINEERING

VALIDATING ARCHITECTURE DECISIONS

Baseline

What the expected service's steady state?
(My boat floats)

Hypothesis

What is the theory you want to validate?
(I believe my boat would still float even if ran into an iceberg)

Experiment

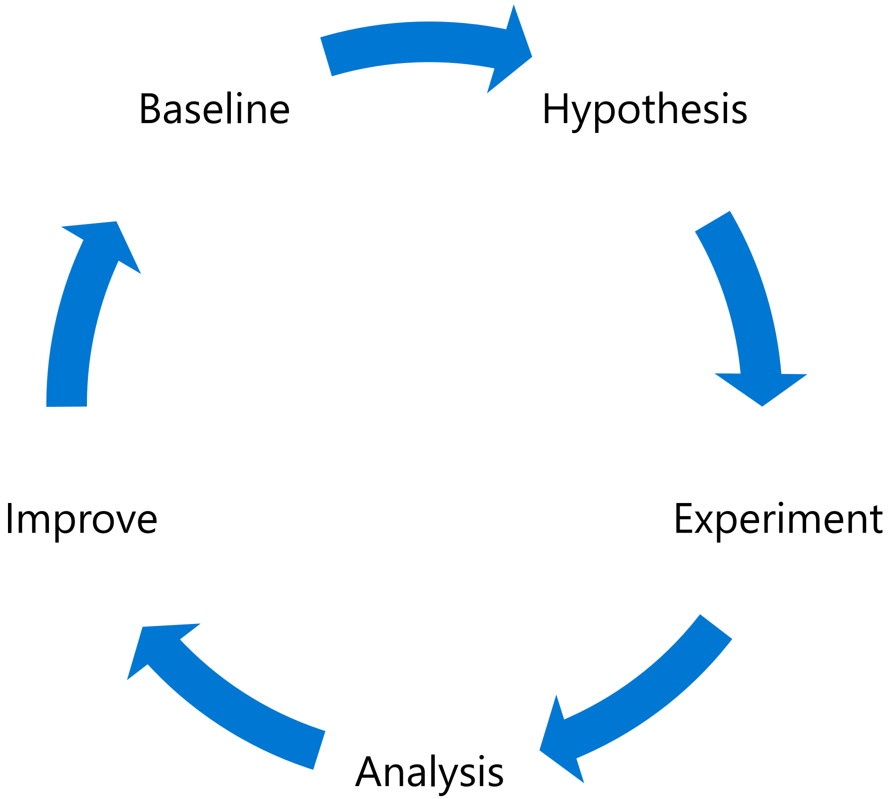
Act to validate the Hypothesis
(I tried running my boat into an iceberg)

Analysis

Validate the response to the experiment
(My boat sank… 😐)

Improve

Take action to address unexpected results
(I have now installed bulkheads on my boat)



Demo

Implementation of **Fault Injection**

UPCOMING IN .NET 8 FOR CLOUD-NATIVE APPLICATION DEVELOPMENT

Building high-scale and high-availability service in .NET 8

Retry Pipelines

`Microsoft.Extensions.Resilience`

`Microsoft.Extensions.Http.Resilience`

- Timeouts
- Retry
- CircuitBreaker
- Bulk Heads
- Rate Limiting
- Hedging

Fault Injections

`Microsoft.Extensions.Resilience.FaultInjection`

- Latency
- Exceptions
- HttpResponse
- CustomResult

subject to change, current .NET 8 RC 1 features

```
// Retry Pipelines in .NET 8
builder.Services.AddGrpcClient<ProductsClient>()
    .AddStandardResilienceHandler();

builder.Services.AddHttpClient<OrderServiceClient>()
    .AddStandardResilienceHandler();


"HttpStandardResilienceOptions": {
  "BulkheadOptions": {
    "MaxConcurrency": 1000,
    "MaxQueuedActions": 0
  },
  "TotalRequestTimeoutOptions": {
    "TimeoutInterval": "00:00:30",
  },
  "RetryOptions": {
    "ShouldRetryAfterHeader": false,
    "RetryCount": 3,
    "BaseDelay": "00:00:02"
  },
  "CircuitBreakerOptions": {
    "FailureThreshold": 0.1,
    "BreakDuration": "00:00:05"
  },
  "AttemptTimeoutOptions": {
    "TimeoutInterval": "00:00:10"
  }
}
```

Bing Image Creator: Little ship on a sunny day, in the middle of the sea, view from above

FOLLOW-UPS

CALL TO ACTION

Today's example: Contonance

- resilient-cloud-apps: Building resilient applications on Azure (github.com)

Azure Reference Architecture & Guidance

- Reliable web app pattern - Azure Architecture Center | Microsoft Learn
- Reliability pillar - Microsoft Azure Well-Architected Framework | Microsoft Learn

.NET 8 Features for Cloud-Native Development

- Cloud-native development with .NET 8 | Microsoft Build 2023 – YouTube
  - Microsoft.Extensions.Http.Resilience Namespace @ .NET 8 RC 1
  - Microsoft.Extensions.Resilience.FaultInjection Namespace @ .NET 8 RC 1

Managed chaos engineering experimentation platform

- Azure Chaos Studio - Chaos engineering experimentation | Microsoft Azure
- Application reliability with Azure Load Testing and Chaos Studio | Microsoft Build 2023

"Don't wait until it crashes. Act now!"

Please provide Session Feedback:

Building resilient applications on Azure

Resilience patterns and validation – a sea of possibilities

https://forms.office.com/r/HK3rbaMZWk