Resilience Engineering

With Polly

verizon media

Trystan Hill

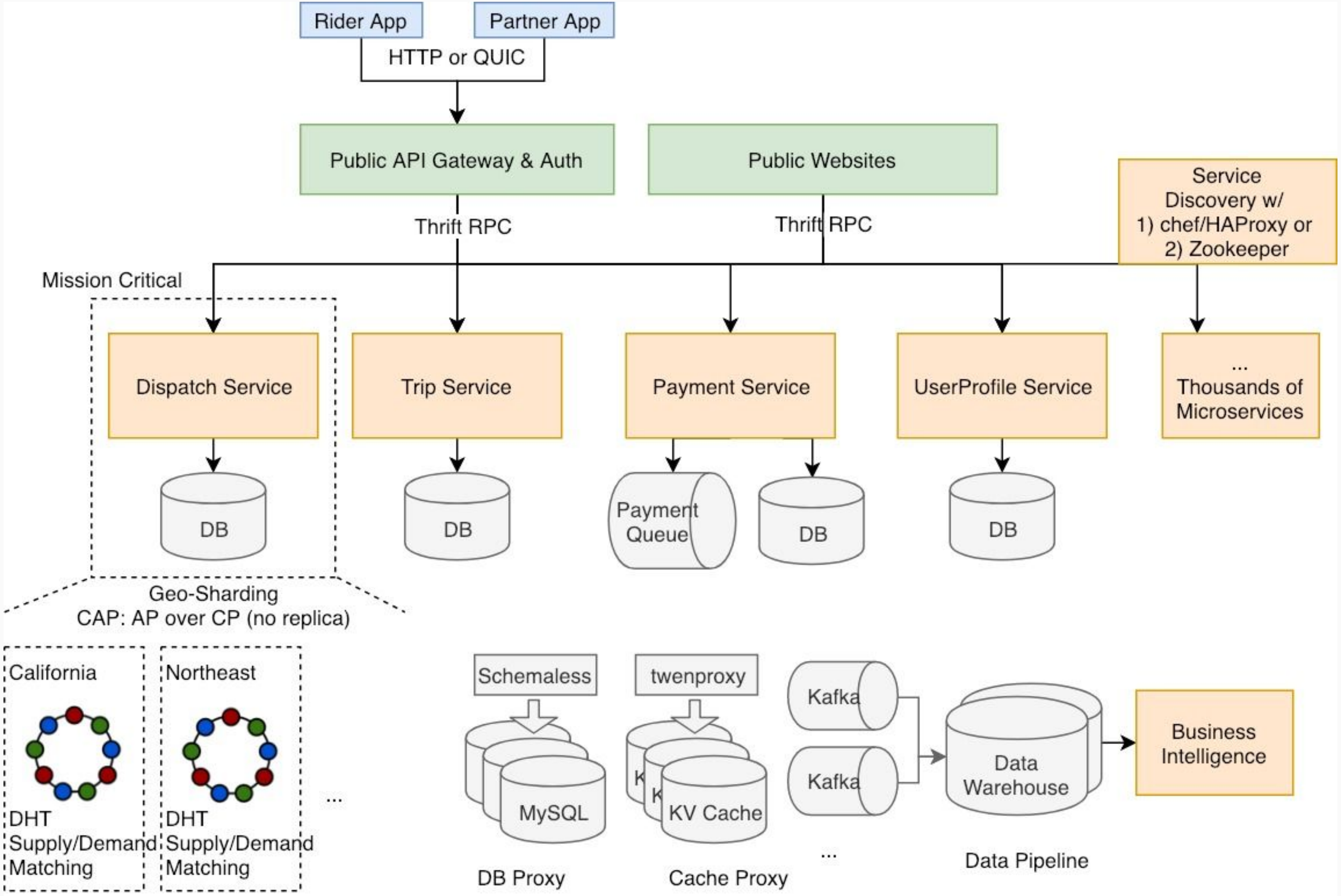
Sr Software Engineer

Shit happens

Systems and services rarely work alone

Systems tend to not have one point of failure, and not in a monolithic app

Failures of various magnitudes happen


Rider App
Partner App
HTTP or QUIC
Public API Gateway & Auth
Public Websites
Service Discovery w/ 1) chef/HAProxy or 2) Zookeeper
Thrift RPC
Thrift RPC
Mission Critical
Dispatch Service
Trip Service
Payment Service
UserProfile Service
... Thousands of Microservices
DB
DB
Payment Queue
DB
DB
Geo-Sharding
CAP: AP over CP (no replica)
California
DHT Supply/Demand Matching
Northeast
DHT Supply/Demand Matching
...
Schemaless
MySQL
DB Proxy
twenproxy
KV Cache
Cache Proxy
Kafka
Kafka
...
Data Warehouse
Data Pipeline
Business Intelligence

Resilience Engineering

Building systems and software that can withstand stressors and still perform core functionality.

e.g. Uber system’s payment system is unreachable for 5 minutes, riders are able to still get rides and are billed once service is restored.

Transient Errors

Network outages

Service outages

Denial of Service attacks

IO locks

Connected device failures

Polly

.NET 4.5+ / .NET Standard 1.1 / .NET Std 2.0+ / .NET Core 2.1+

Fluently express transient exception handling policies in a thread safe manner

https://github.com/App-vNext/Polly

Nuget: Install-Package Polly

Resiliency Strategies / Policies

Retry … ‘Maybe it’s just a blip’

Timeout … ‘Don’t wait forever!’

Circuit Breaker … ‘That system is down / struggling’

Bulkhead isolation … ‘One fault shouldn’t sink the whole ship’

Cache … ‘You’ve asked that one before!’

Fallback … ‘If all else fails … degrade gracefully’

All policies can be combined, for multiple protection!

Retry

Many faults are transient and may self-correct after a short delay.

```
for (int retryAttempt = 0; retryAttempt < MaxRetries; retryAttempt++) {
   response = await httpClient.SendAsync(request, cancellationToken);
   if (response.IsSuccessStatusCode) {
       return response;
   }
   await Task.Delay(TimeSpan.FromMilliseconds(10));
)
```

Retry (cont.)

Exponential delay strategies

```
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(new[]
  {
    TimeSpan.FromSeconds(1),
    TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(4),
    TimeSpan.FromSeconds(8),
    TimeSpan.FromSeconds(15),
    TimeSpan.FromSeconds(30)
  });
```

```
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(3, retryAttempt =>
    TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt))
    );
```

```
Random jitterer = new Random();
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(5,
      retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
                    + TimeSpan.FromMilliseconds(jitterer.Next(0, 1000))
  );
```

So you're telling me there's a chance.

Timeout

Beyond a certain wait, a success result is unlikely.

```
Policy
  .Timeout(30, onTimeout: (context, timespan, task) =>
    {
        logger.Warn($"execution timed out after {timespan.TotalSeconds} seconds.");
    });
```

Timeout (cont.)

Two modes:

`TimeoutStrategy.Optimistic`

For delegates that support & respect CancellationTokens

`TimeoutStrategy.Pessimistic`

For delegates that don’t and might need forceful termination

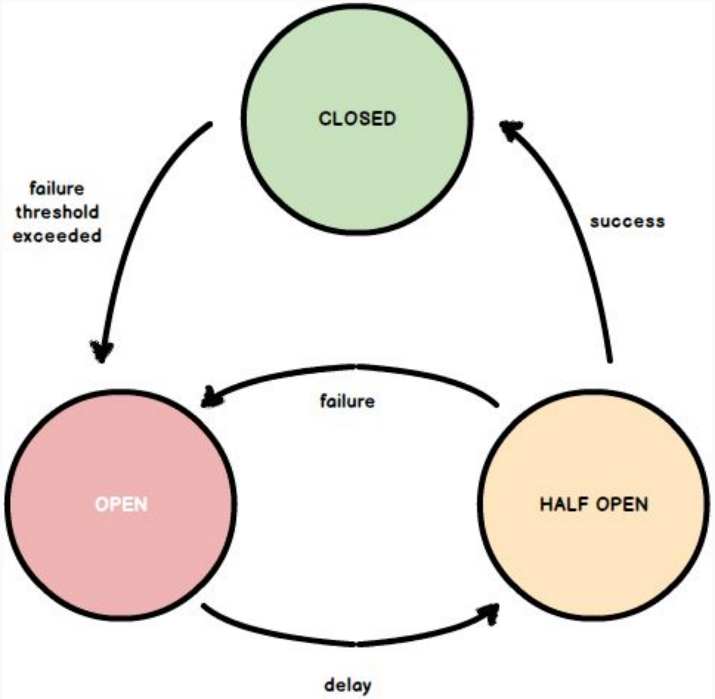
Circuit-breaker

When a system is seriously struggling, failing fast is better than making users/callers wait. Protecting a faulting system from overload can help it recover.

The analogy of an electrical circuit with three states:

- Closed: calls flow
- Open: calls fail immediately
- Half-open: for determining circuit health

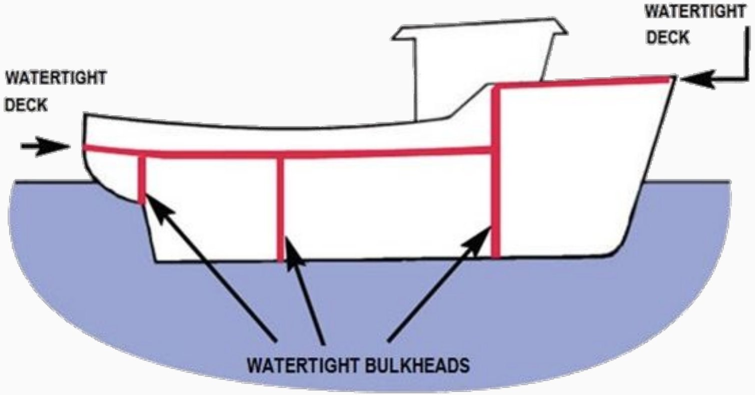
Circuit-breaker (continued)



THE CALLS TO DEPENDENT SERVICES
MUST FLOW

Bulkhead Isolation

- Control of concurrency
- Conceptually like a “thread pool” or “worker pool”



Bulkhead Isolation (cont.)

```
// Restrict executions through the policy to a maximum of twelve concurrent actions,
// with up to two actions waiting for an execution slot in the bulkhead if all slots are
taken.
var bulkhead = Policy.Bulkhead(12, 2,
     onBulkheadRejected: _ => { _logger.LogInformation("Max parallelism reached");
});
// ...
int freeExecutionSlots = bulkhead.BulkheadAvailableCount;
int freeQueueSlots     = bulkhead.QueueAvailableCount; //keep value low for sync
```

- BulkheadRejectedException thrown once max parallelism is reached
- This can be a good time to horizontally scale (via either the queue metrics or using the OnBulkheadRejected / OnBulkheadRejectedAsync delegate.
- Configuring max parallelism will be different for IO vs CPU bound work

Fallback

Provide a default or “fallback” value if execution fails

```
Policy<UserAvatar>
   .Handle<FooException>()
   .Fallback<UserAvatar>(UserAvatar.Blank, onFallback: (exception, context) =>
    {
        logger.Warn($"Default avatars again :(", exception);
    });
```

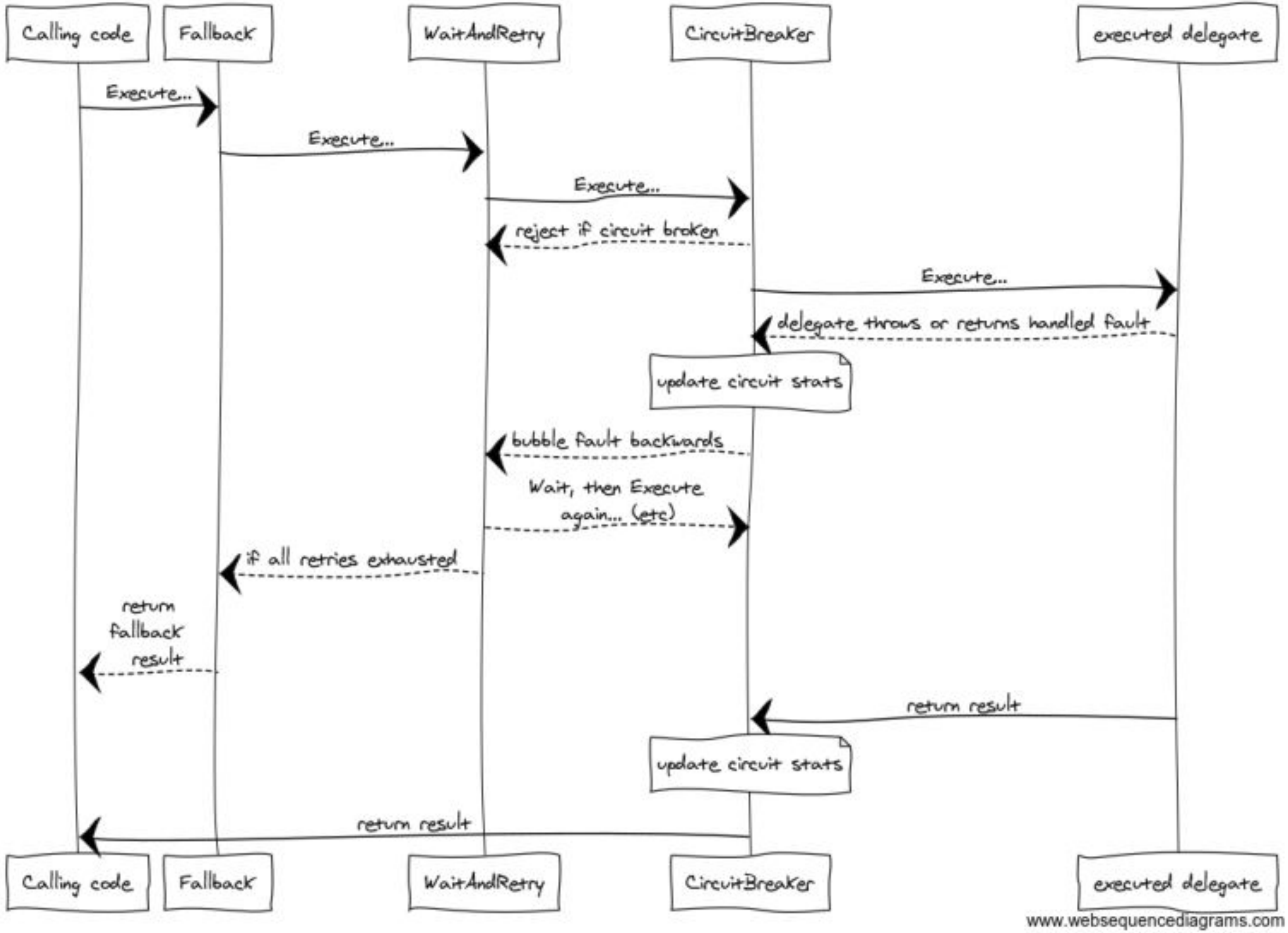
Combining Multiple Policies

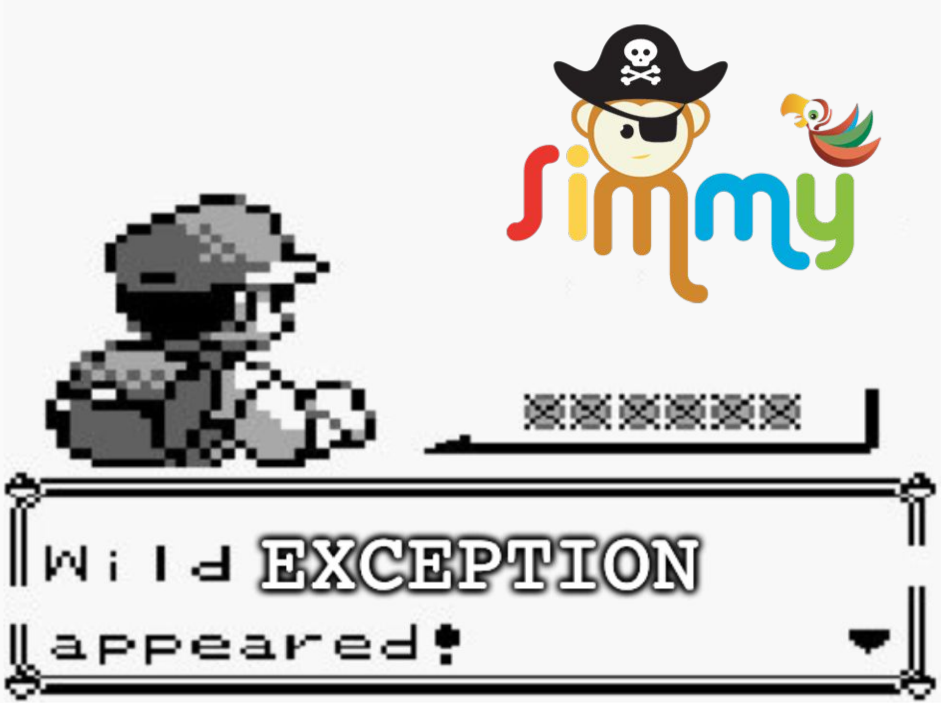
```
var redisFallbackPolicyString = Policy<string>.Handle<RedisConnectionException>()
   .Or<SocketException>()
   .FallbackAsync(
       fallbackValue: null,
       onFallbackAsync: async b => {
           await Task.CompletedTask;
           _logger.Log(LogLevel.Error, $"Fallback'");
           return; });

var retryPolicy = Policy.Handle<RedisConnectionException>()
   .WaitAndRetryAsync(retrycount, retryAttempt => TimeSpan.FromSeconds(1), (excp, ts, retryCount, context) =>{
       logger.Log(LogLevel.Error, $"Redis error on retry {retryCount} for {context.PolicyKey}", exception);
   });

var circuitBreakerPolicy = Policy.Handle<RedisConnectionException>()
   .CircuitBreakerAsync(
       exceptionsAllowedBeforeBreaking: 3,
       durationOfBreak: TimeSpan.FromSeconds(120),
       onHalfOpen: () => { logger.LogInfo($"Redis caching circuit breaker half open"); },
       onBreak: (exception, ts) => { logger.LogInfo($"Circuit breaker open for {ts.TotalSeconds}"); },
       onReset: () => { logger.LogInfo($"Redis caching circuit breaker: closed"); }
);

var pol = redisFallbackPolicyString.WrapAsync(retryPolicy).WrapAsync(circuitBreakerPolicy);
```


Calling code
Fallback
WaitAndRetry
CircuitBreaker
executed delegate
Execute..
Execute..
Execute..
reject if circuit broken
Execute..
delegate throws or returns handled fault
update circuit stats
bubble fault backwards
Wait, then Execute again... (etc)
if all retries exhausted
return fallback result
return result
update circuit stats
return result
Calling code
Fallback
WaitAndRetry
CircuitBreaker
executed delegate
www.websequencediagrams.com


simmy
Wild EXCEPTION
appeared!

Chaos Monkey / Engineering

Level 1: Inject faults into integration tests

Level 2: Inject faults into regression / QA tests

Level 3: PRODUCTION MONKEY

Polly Simmy

- Exception: Injects exceptions in your system.
- Result: Substitute results to fake faults in your system.
- Latency: Injects latency into executions before the calls are made.
- Behavior: Allows you to inject any extra behaviour, before a call is placed.

https://github.com/Polly-Contrib/Simmy

Onto the demo...

Leftover junk

HttpClientFactory HttpClient and Polly integration

Polly, HttpClientFactory and the Policy Registry in a console application

HTTPCLIENTFACTORY IN ASP.NET CORE 2.1

Further Polly Simmy Chaos :

Polly.Contrib.SimmyDemo_WebApi

Simmy, the monkey for making chaos

Try .NET Samples of Polly & Simmy