Building resilient applications on Azure

27. April 2022

CLOUD NATIVE
COMPUTING FOUNDATION

Microsoft Azure

Microservices Serverless Containers

Presenting today

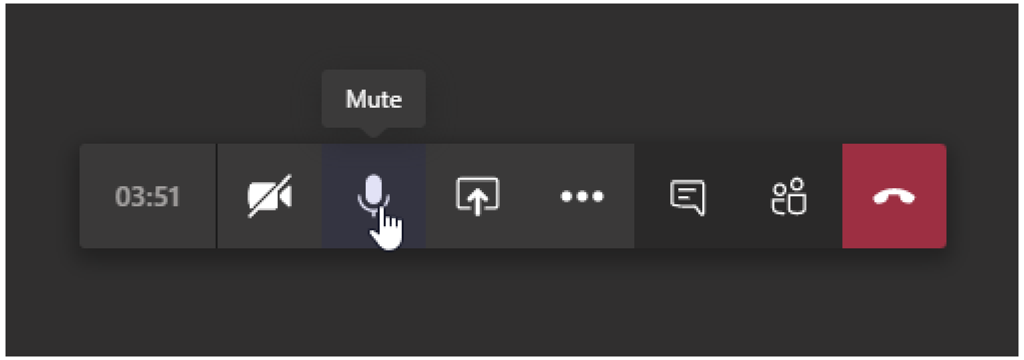
Ricardo Niepel
Cloud Solution Architect – Azure App Dev
ricardo.niepel@microsoft.com
RicardoNiepel
@RicardoNiepel

Dennis Zielke
Technical Lead - Cloud Native Apps
dennis.zielke@microsoft.com
denniszielke
@denzielke

Organizational Upfront

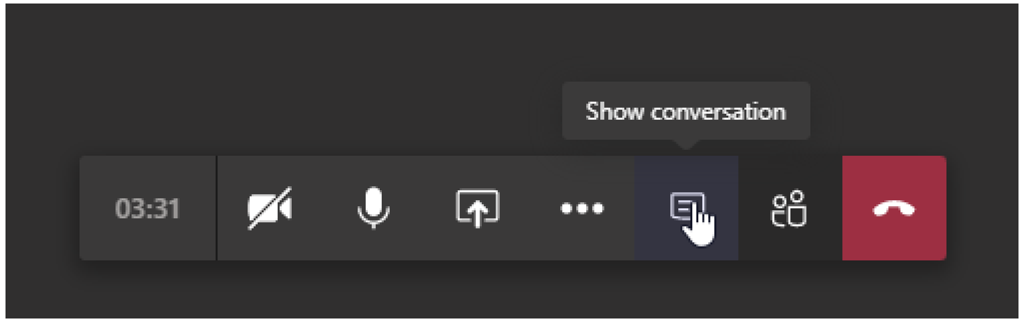
- Schedule: Monthly, 2nd or 3rd Friday
- If your colleagues have trouble joining this Teams call:

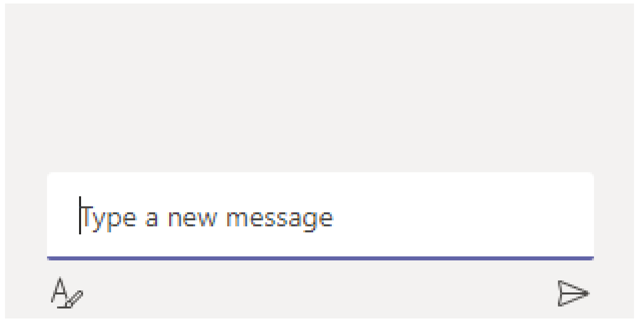
  Use the Web Client (no plug-in) - Google Chrome version >72 or Microsoft Edge (Chromium-based)
- No recordings / if you miss it pls join next time
- Please make sure to mute yourself, too many people



Questions

If you have any Cloud Native relevant question, please use the Teams chat



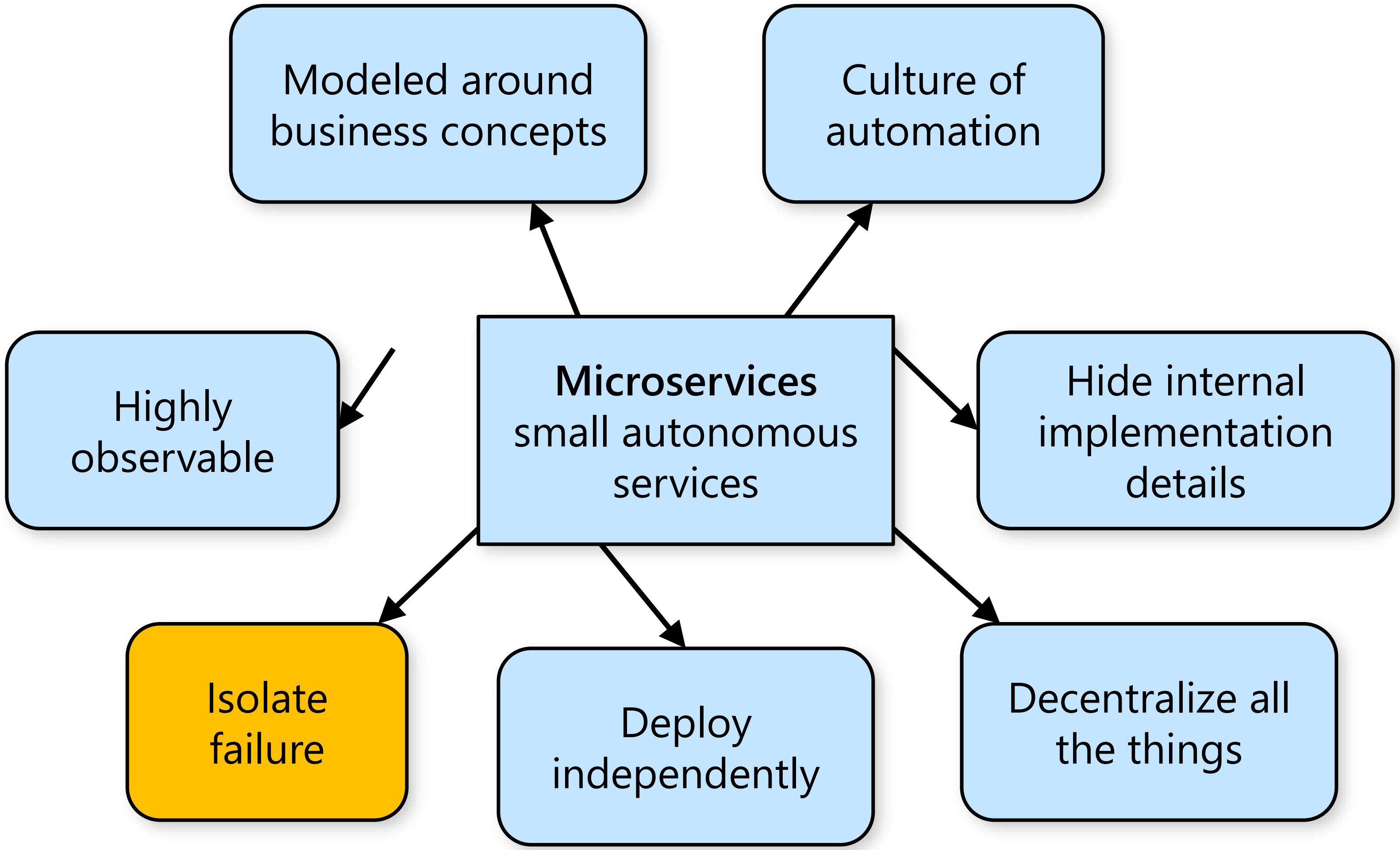


If we have time left at the end, we will pick some questions and answering them live

Agenda for today

- How can cloud native applications fail?
- How can you build your appliation resilient to failure?
- How can I implement a resilient appliation architecture?
- How can you validate your assumptions?

Principles of microservices


Microservices small autonomous services
Modeled around business concepts
Culture of automation
Highly observable
Hide internal implementation details
Isolate failure
Deploy independently
Decentralize all the things

Sam Newman, Building Microservices

Microsoft Azure

Reliability and Resilience

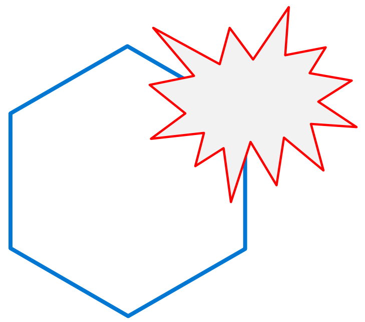
Reliable workloads are:

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

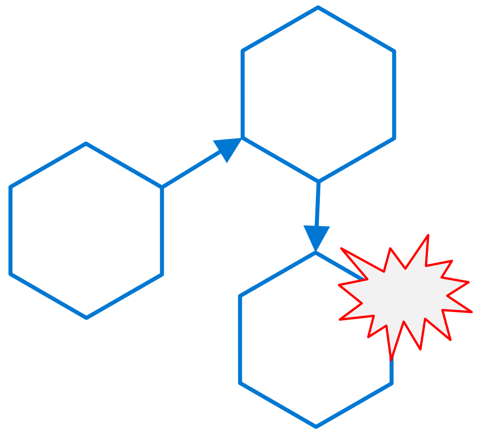


Focus on: Minimizing risk of failure

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?

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 infrastructure resources

Partial infrastructure failures are more common

Health probe unreliability

Service works but 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

Traits of Resilient Systems

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.

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.

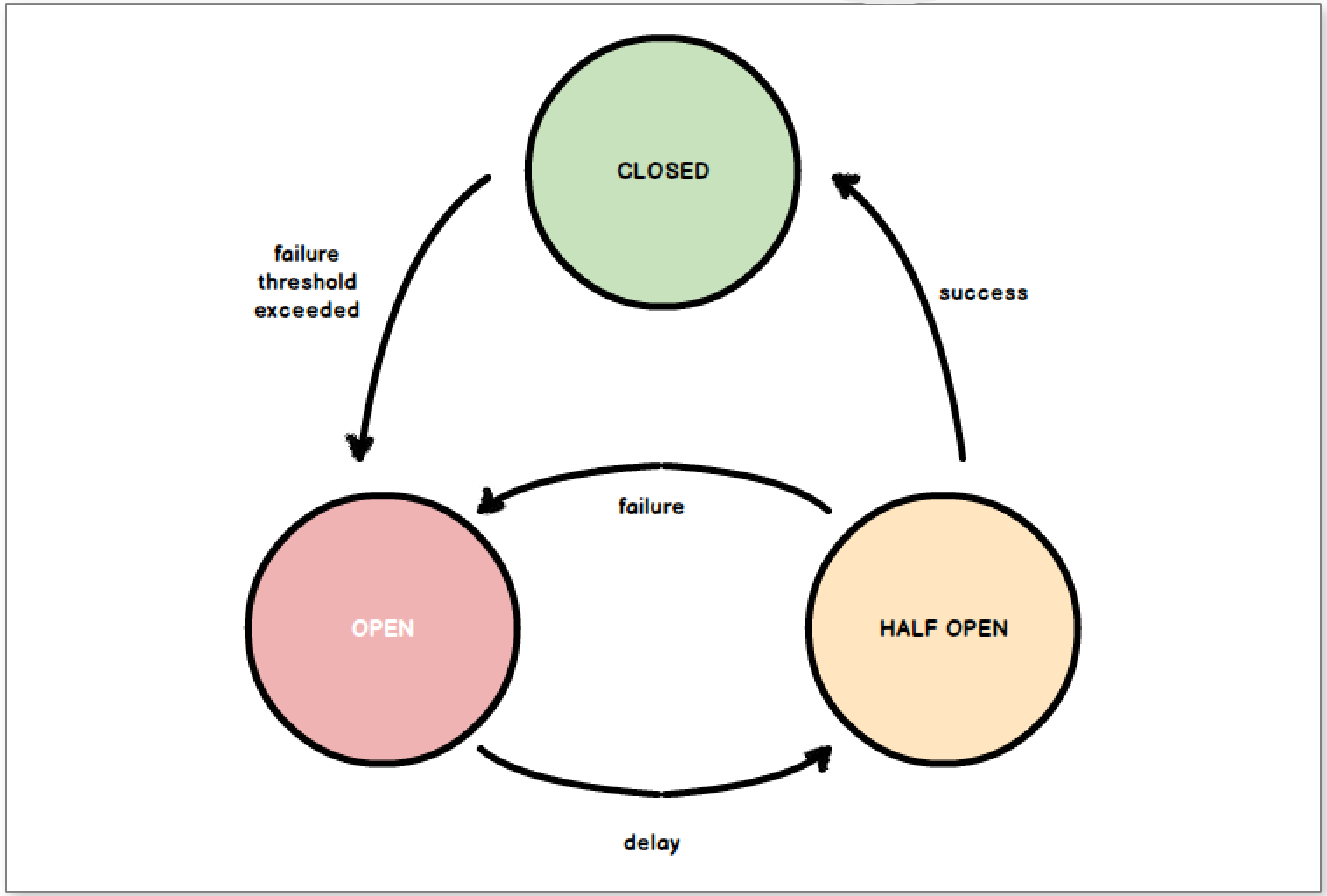
If the problem appears to have been fixed, the application can try to invoke the operation.

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.



Timeout - *Don't wait forever.*

Solution

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

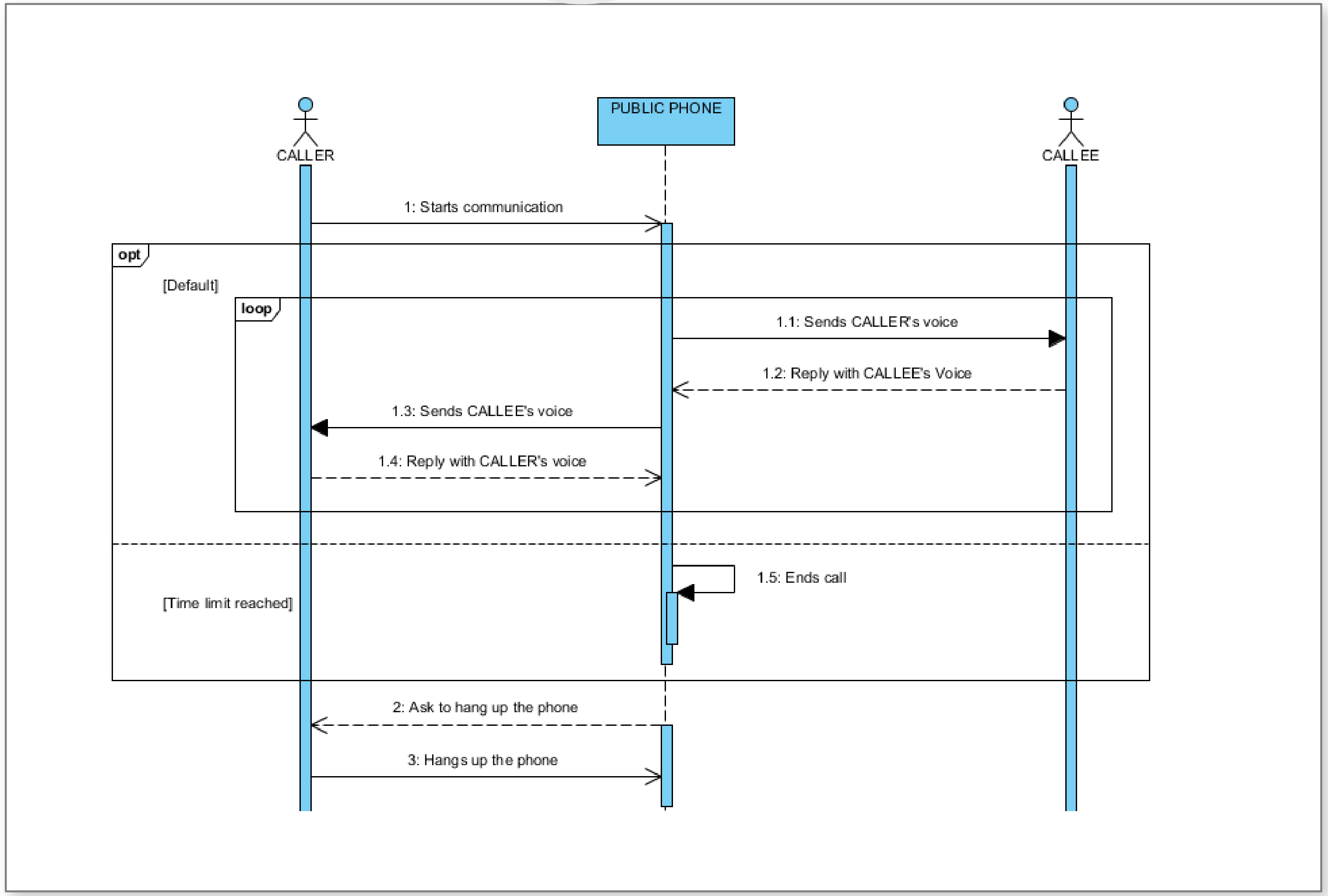
Waiting forever (having no timeout) is a bad design strategy. It leads to the blocking up of threads or connections (itself often a cause of further failure), during a faulting scenario.

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.


CALLER
PUBLIC PHONE
CALLEE
1: Starts communication
opt
[Default]
loop
1.1: Sends CALLER's voice
1.2: Reply with CALLEE's Voice
1.3: Sends CALLEE's voice
1.4: Reply with CALLER's voice
1.5: Ends call
[Time limit reached]
2: Ask to hang up the phone
3: Hangs up the phone

Retry - *Maybe it's just a blip!*

Health Probe failure

Solution

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

Considerations

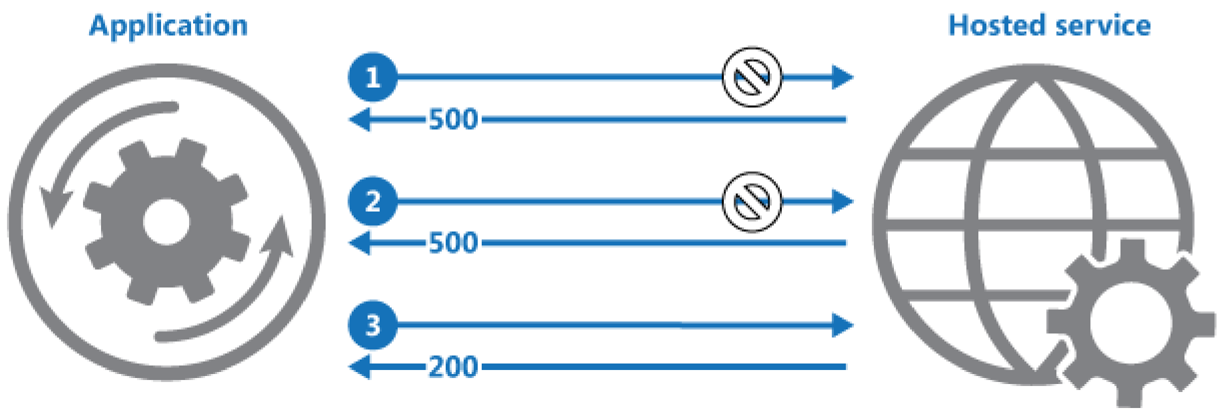
Only works if you can identify transient errors.

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 of load.

Beware

Only retry idempotent operations or provide compensating logic.



1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Throttling - *Enough is enough.*

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.

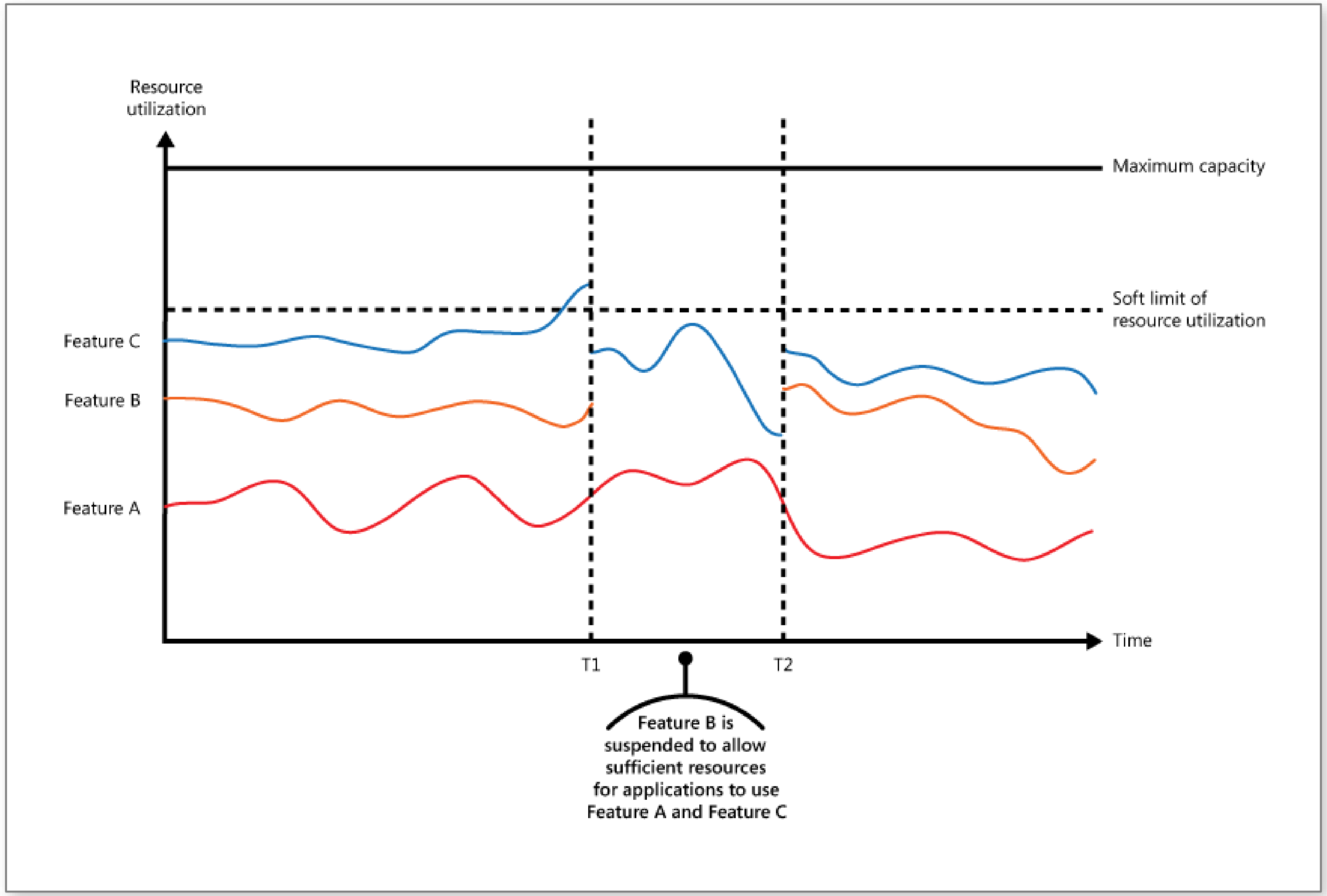
Considerations

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!

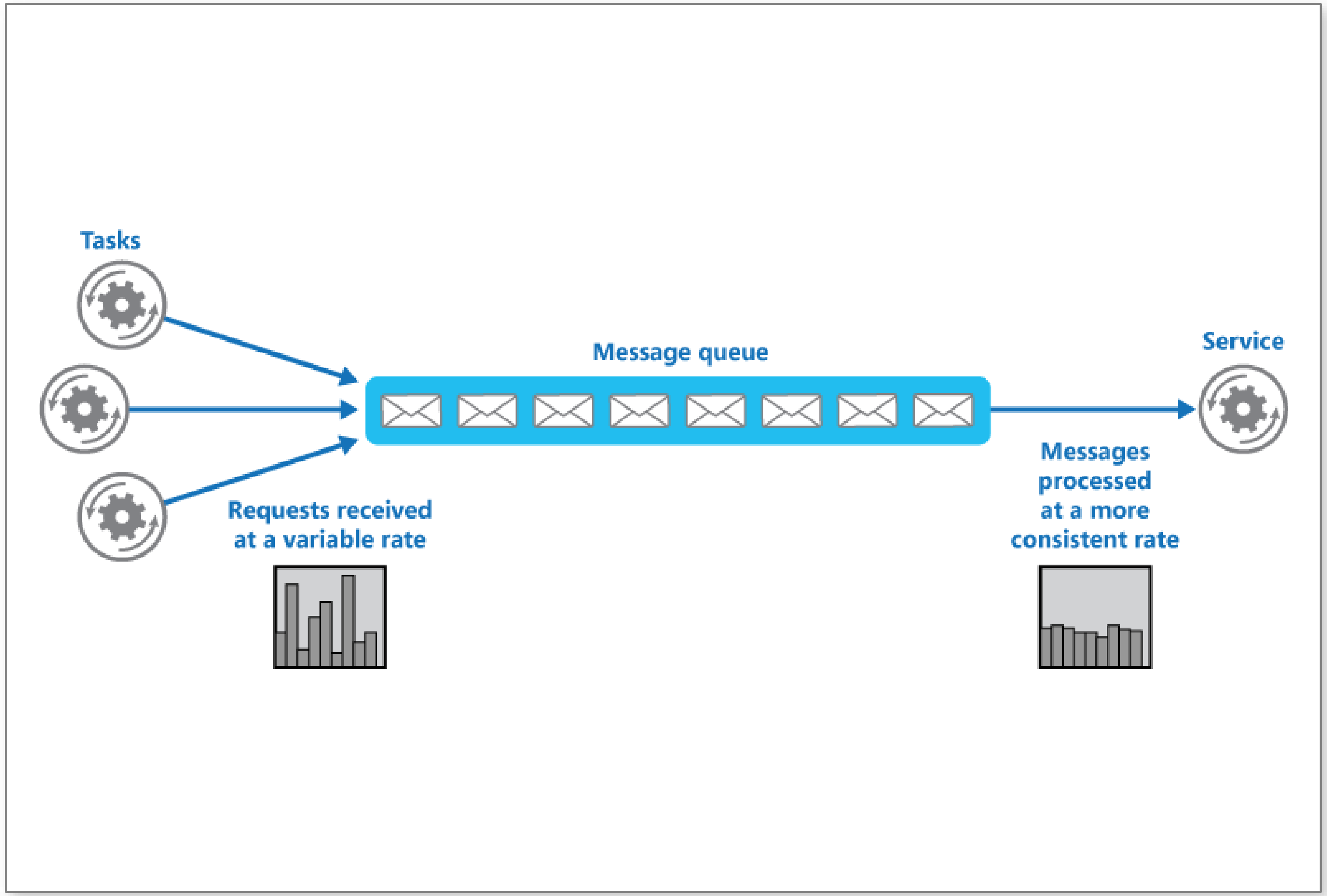


Queue Load Leveling - *Keep calm and dispatch a message.*

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.


Tasks
Message queue
Service
Requests received at a variable rate
Messages processed at a more consistent rate

Microsoft Azure

Thank you.

Slides & FAQ: