Cloud First Architecture

Patterns and Practices for the Cloud

WELCOME

Cameron Vetter

Cloud / Machine Learning / Mixed Reality Consultant

Cameron Vetter is a technologist with 20 years of experience using Microsoft tools and technologies to develop software. Cameron has experience in many roles including Development, Architecture, Infrastructure, Management, and Leadership roles. He recently received a Microsoft MVP award for his evangelism work around Deep Learning in Azure. He has worked for some of the largest companies in the world as well as small companies getting a breadth of experience helping him understand the needs of different size businesses and different Industries. Currently, Cameron is the Principal Architect at the Octavian Technology Group, where he helps clients develop Technical Strategies. He also helps clients Architect, Design, and Develop software focusing on Deep Learning / AI, Cloud Architecture, Microservices, Mixed Reality, and Azure.

About Us

A Partner to Advise and Support

Our team offers a combined decades of experience in technology-related fields, and we leverage our expertise to take a business-focused approach to helping organizations solve real problems with proven solutions.

Octavian TG offers Cloud Architecture, Mixed Reality Development, Data Science, Machine Learning, Fractional CIO, and Agile trainers.

OCTAVIAN TECHNOLOGY GROUP

Why Azure?

Everything we talk about today can be applied to any major cloud providers' offerings. I use Azure as example, because they have the most sophisticated offering and I have the most familiarity with it.

Credit: Azure Architecture Center @ https://docs.microsoft.com/en-us/azure/architecture/

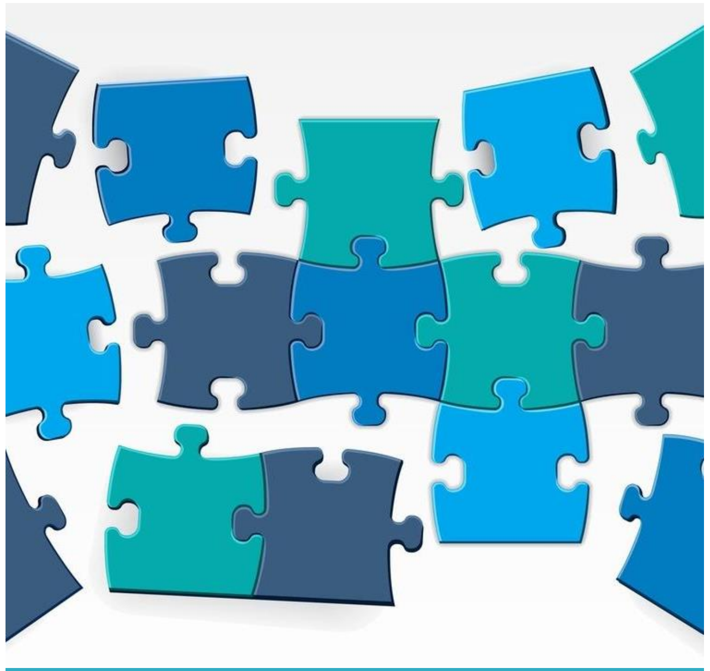
Agenda

Hosting Model

Cloud Hosting Models

/ Good, Better, Best /

Infrastructure AAS



Platform AAS

Functions AAS

INFRASTRUCTURE AS A SERVICE

/ Good/

ADVANTAGES	DISADVANTAGES
✓ Servers. Network, and Data Center management managed by Cloud Provider.	✗ Operating System Updates, Application installation,, and database server management unchanged.
✓ Availability Sets allow duplicate VM's to exist in different data centers for scaling.	✗ Security is completely dependent on proper configuration of the plafform.
✓ Closest to on premise, allowing for lift and shift migrations.	✗ Lift and shift migrations usually reproduce most of your problems in a new location.

PLATFORM AS A SERVICE

/ Better /

ADVANTAGES

- ✓ All levels of the infrastructure are managed by the cloud provider
- ✓ Most security is handled by the cloud providers security team.
- ✓ Automatic scaling and replication is available.

DISADVANTAGES

- ✗ Application and Service change management is unchanged.
- ✗ Application and Services are responsible for not opening up security holes.
- ✗

FUNCTIONS AS A SERVICE

/ Best /

ADVANTAGES

- ✓ Creation of compute is handled by the platform, no need to worry about selecting the right resources.
- ✓ Extremely cost efficient.
- ✓

DISADVANTAGES

- ✗ Lack of flexibility, software must be designed with a SOA pattern.
- ✗ Limited to the FAAS platforms selection of tools and languages.
- ✗

Lift and Shift

A strategy for migrating a workload to the cloud without redesigning the application or making code changes. Sometimes called rehosting.

Cloud Optimized

A strategy for migrating to the cloud by refactoring an application to take advantage of cloud-native features and capabilities.

Architecture Styles

Architectures

Dependency Management Appropriate for your Domain Type

Traditional Business Domain / Few Updates

N-tier

Horizontal tiers

Simple Domain / Resource Intensive

Web-Queue-Worker

Front and Backend jobs decoupled with async messaging

IoT and real-time systems / Frequent Updates

Event Driven

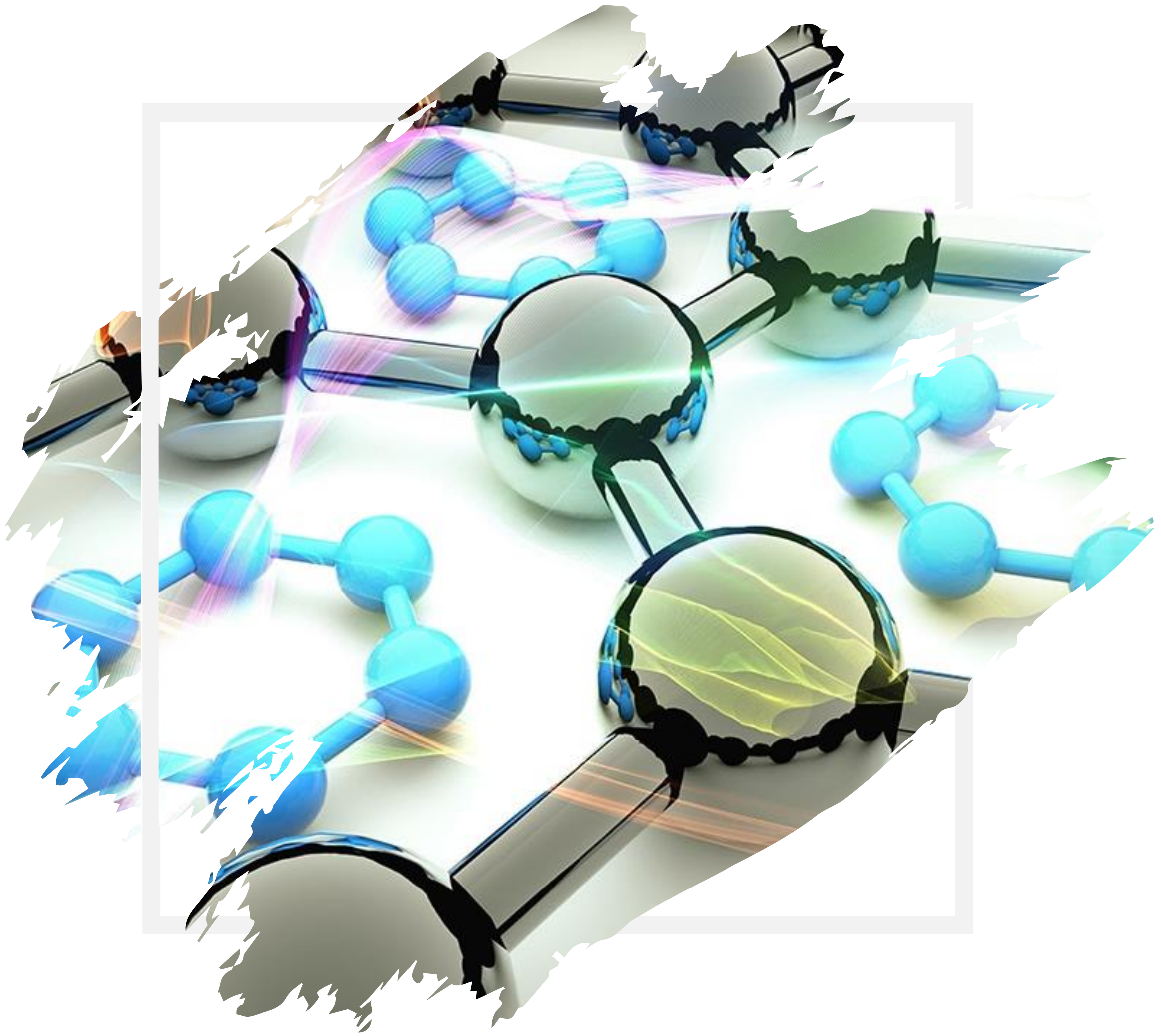
Producer / Consumer. Each subsystem is independent.

Complicated Domain / Frequent Updates

Microservices

Vertically decomposed services that interact through an API

Design Principles

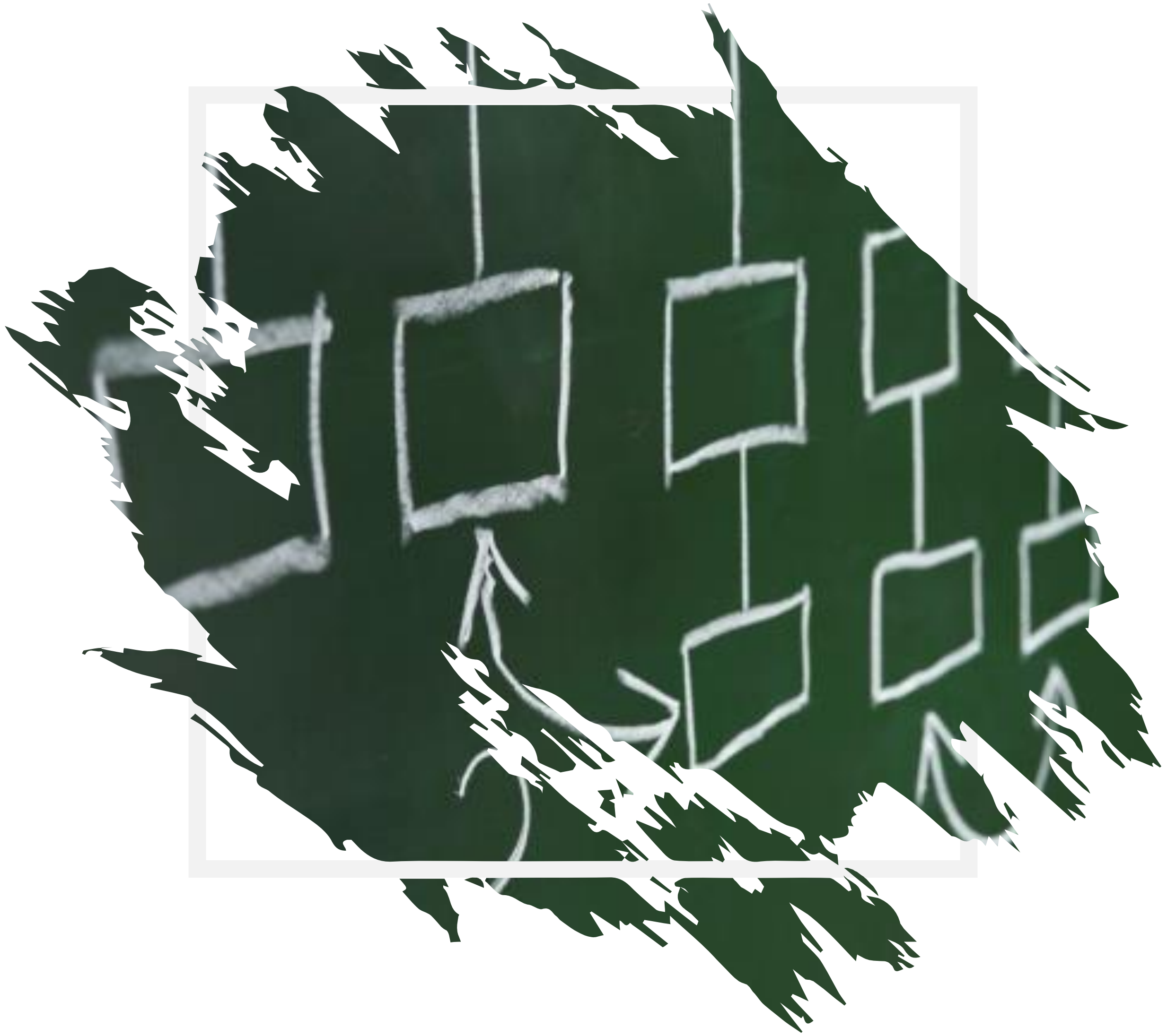


DESIGN FOR SELF HEALING

In a distributed system, failures happen. Design your application to be self healing when failures occur.

MAKE EVERYTHING REDUNDANT

Build redundancy into your application, to avoid having single points of failure.



MINIMIZE DEPENDENICES

Minimize dependencies between application services to achieve scalability.

DESIGN TO SCALE OUT

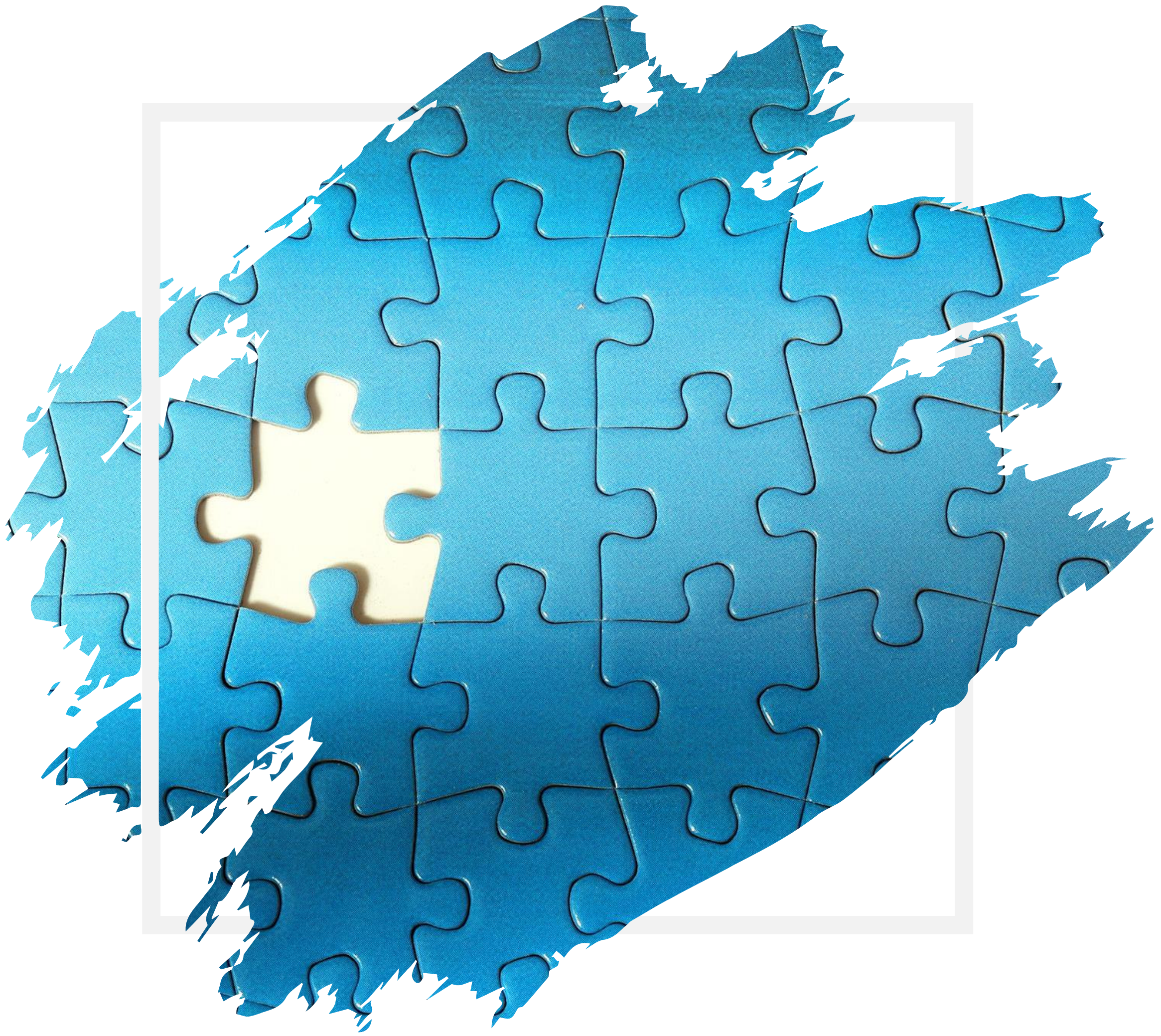
Design your application so that it can scale horizontally, adding or removing new instances as demand requires.

PARTITION AROUND LIMITS

Use partitioning to work around database, network, and compute limits.

DESIGN FOR OPERATIONS

Design your application so that the operations team has the tools they need.



USE MANAGED SERVICES

When possible, use platform as a service (PaaS) rather than infrastructure as a service (IaaS).

USE THE BEST DATA STORE FOR THE JOB

Pick the storage technology that is the best fit for your data and how it will be used.

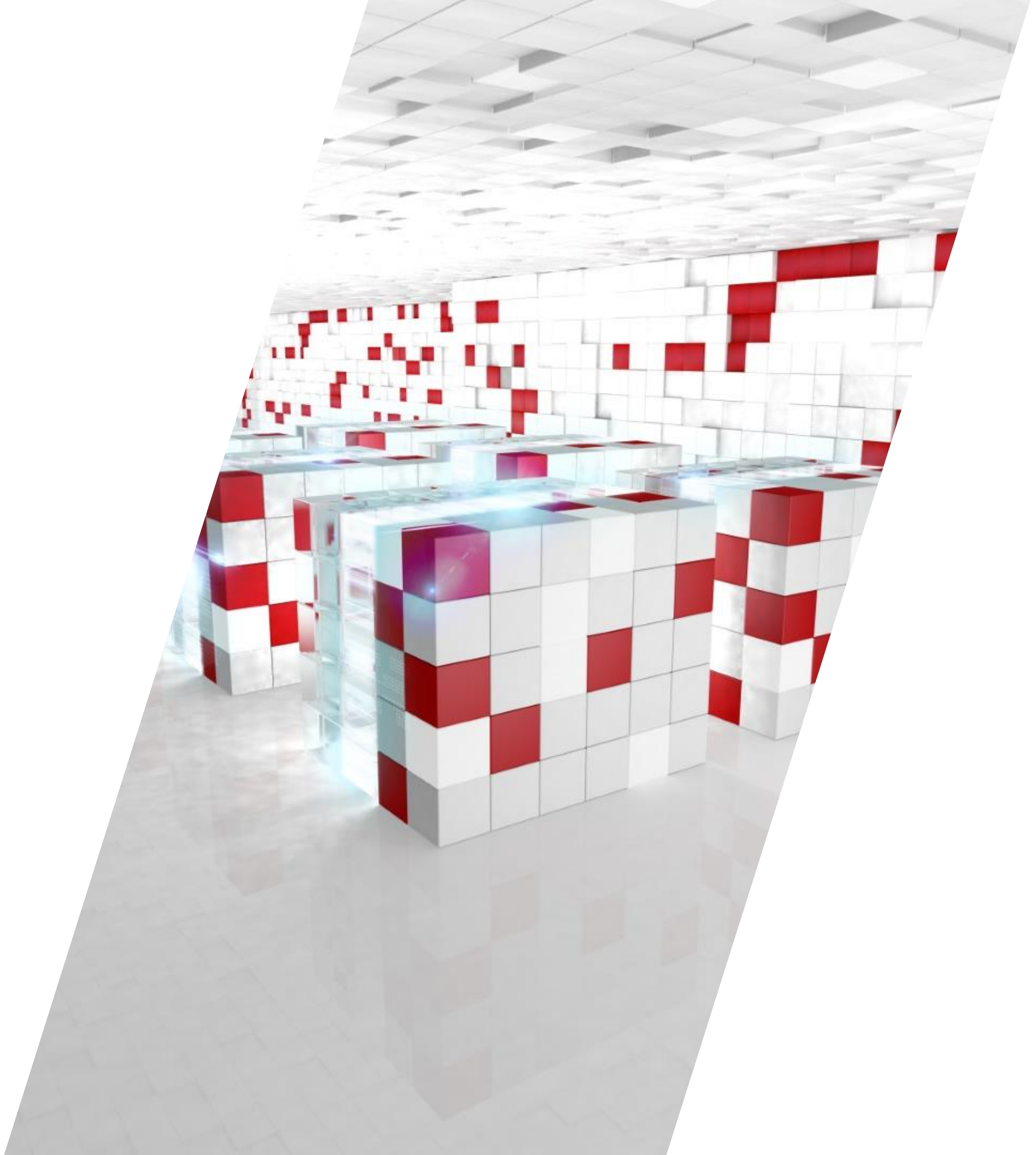
DESIGN FOR EVOLUTION

All successful applications change over time. An evolutionary design is key for continuous innovation.

BUILD FOR THE NEEDS OF THE BUSINESS

Every design decision must be justified by a business requirement.

Best Practices



Scalability

Scalability is the ability of a system to handle increased load. There are two main ways that an application can scale. Vertical scaling (scaling *up*) means increasing the capacity of a resource, for example by using a larger VM size. Horizontal scaling (scaling *out*) is adding new instances of a resource, such as VMs or database replicas.

The need to vertically scale signifies a problem.

Availability

Availability is the proportion of time that the system is functional and working. It is usually measured as a percentage of uptime. Application errors, infrastructure problems, and system load can all reduce availability.

SLA is a combined effort of the cloud provider and the application architecture

Resiliency

Resiliency is the ability of the system to recover from failures and continue to function. The goal of resiliency is to return the application to a fully functioning state after a failure occurs. Resiliency is closely related to availability.

A system with poor availability has a problem with resiliency

Management and DevOps

Deployments must be reliable and predictable. They should be automated to reduce the chance of human error. Monitoring and diagnostics are crucial.

Failed deploys usually are a symptom of a problem with your DevOps

Security

You must think about security throughout the entire lifecycle of an application, from design and implementation to deployment and operations. The platform should provide protections against a variety of threats, but you still need to build security into your application and into your DevOps processes.

Don’t wait to start thinking about security until after your first breech.

Cloud Design Patterns

Design Patterns

Anti-Corruption Layer

Implement a façade or adapter layer between a modern application and a legacy system

Design Patterns

Backends for Frontends

Create separate backend services to be consumed by specific frontend applications or interfaces.

Design Patterns

Circuit Breaker

Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.

CLAIM CHECK
IN CASE OF ANY LOSS, CLAIM BEFORE LEAVING
Not Responsible for Contents of Garments or anything left overnight
001

Design Patterns

Claim Check

Split a large message into a claim check and a payload to avoid overwhelming a message bus.

Design Patterns

Competing Consumers

Enable multiple concurrent consumers to process messages received on the same messaging channel.

Design Patterns

Federated Identity

Delegate authentication to an external identity provider.

Design Patterns

Publisher/Subscriber

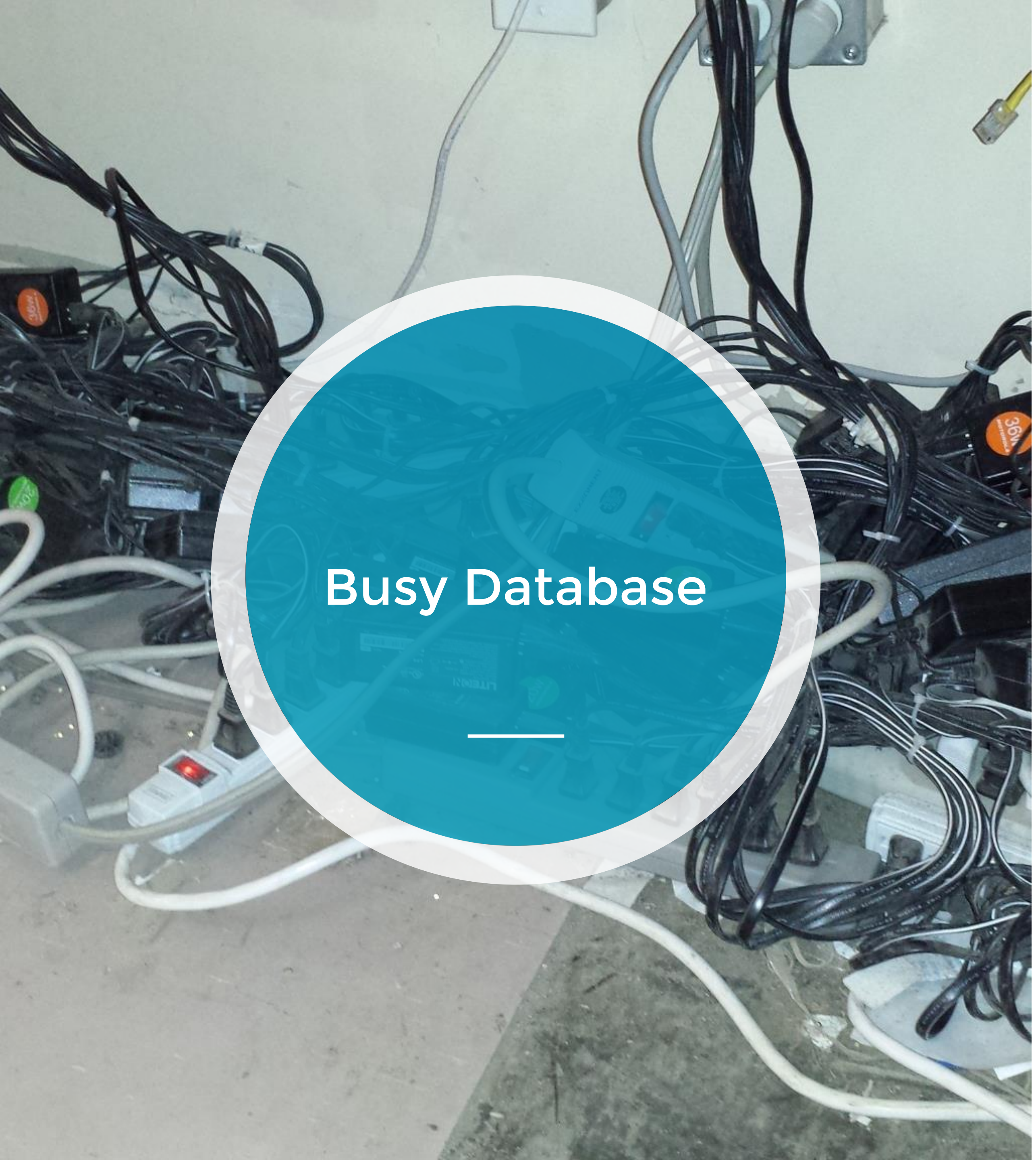
Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.

Design Patterns

Strangler

Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

Antipatterns



Busy Database

PROBLEM

Database Code execution, such as stored procedures and triggers overused, putting excessive load on the database server.

DETECTION

Monitor the volume of database activity, compare it to the usage of the other tiers.

SOLUTION

Refactor processing to other application tiers, limiting your database to data access operations..

PROBLEM

Long synchronous tasks or excessive background threads can cause decreased response times

DETECTION

High latency on front end tasks and server failures including 500 or 503 errors.

SOLUTION

Make all front end tasks asynchronous and move resource intensive tasks to isolated compute.

Busy Front End

Chatty I/O

PROBLEM

A high quantity of network calls and other I/O operations like disk operations.

DETECTION

End users report extended response times or failures caused by timeouts, due to resource contention.

SOLUTION

Reduce the quantity of I/O requests by batching data into larger requests.

PROBLEM

Application retrieves lots more data than it needs which often gets discarded. Improper use of ORM tools to filter data retrieval in memory.

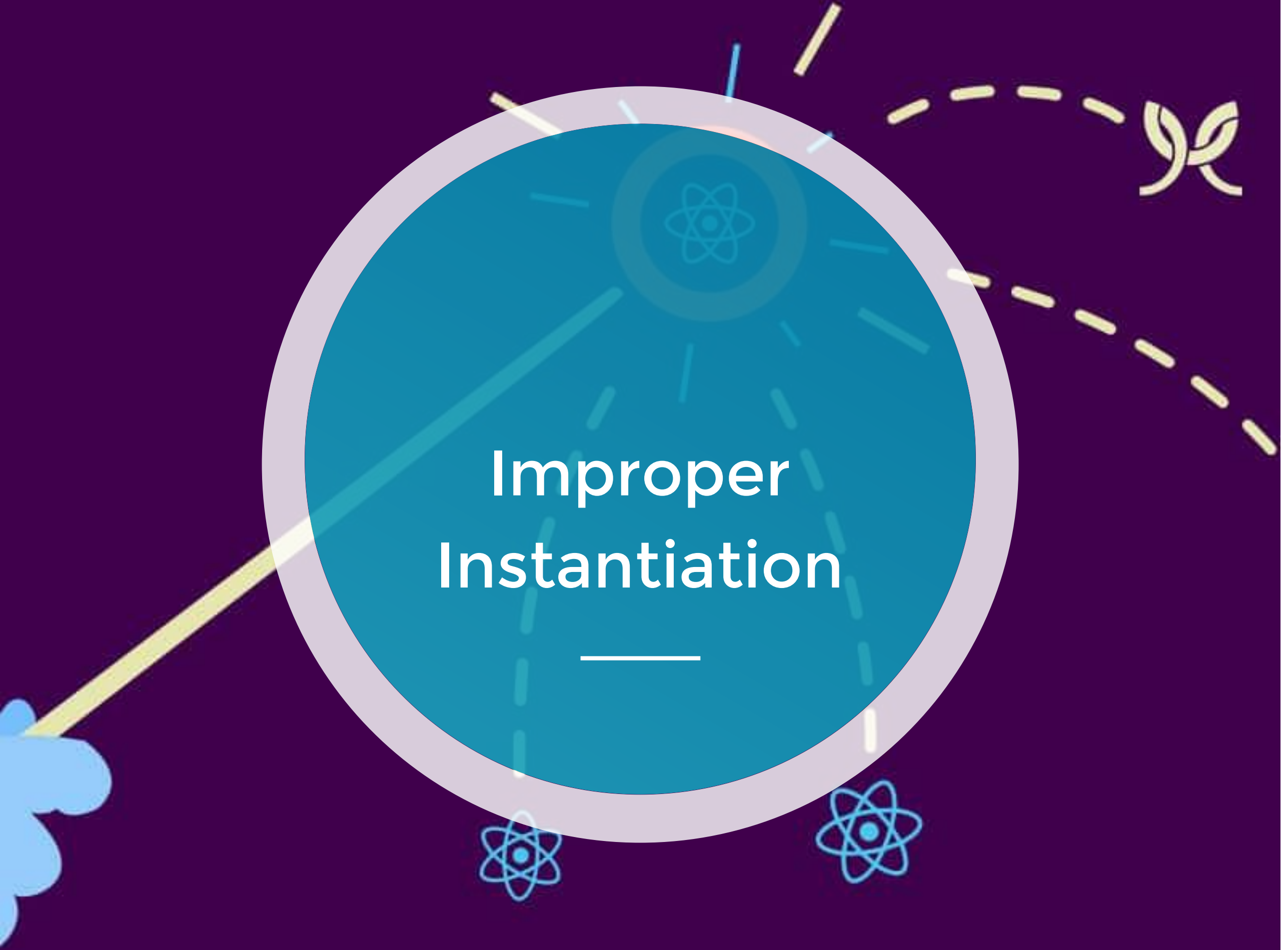
DETECTION

High latency and data store contention, Long running queries are identified.

SOLUTION

Fetch only the data that you need. Optimize ORM based requests to filter data at the database server not in memory.

Extraneous Fetching



Improper Instantiation

PROBLEM

Using the wrong instantiation lifetime for classes, Not using a Singleton pattern where appropriate.

DETECTION

Exceptions related to exhaustion of resources, increased memory usage and garbage collection.

SOLUTION

Wrap classes in thread safe singleton's when they are safe for reuse. Use resource pooling when appropriate.

PROBLEM

Putting all of an applications data into a single data store, that may lead to resource contention or be a poor fit for some of the data.

DETECTION

Sudden dramatic slow downs that lead to eventual failures.

SOLUTION

Separate data according to use aligned with how the data is used.

Monolithic Persistence

PROBLEM

Repeatedly fetching the same information from a resource that is expensive. Repeatedly constructing the same calls to a remote service.

DETECTION

Exceptions related to exhaustion of resources, increased memory usage and garbage collection.

SOLUTION

Reads should check a cache then retrieve the data if it is not cached.

Any Questions?

www.cameronvetter.com

@poshporcupine

Linkedin.com/in/cameronvetter