Cloud Native Architecture Patterns

*Patterns should...be described uniformly.*

*This helps us to compare one pattern with another...*

*Pattern-Oriented Software Architecture*

*We describe design patterns using a consistent format...making design patterns easier to learn, compare, and use.*

*Design Patterns: Elements of Reusable Object-Oriented Software*

## Brick and Mortar Pattern Template

Context

*The basic situation in which we find ourselves working.*

Problem

*Presents the problem as a system forces which must be balanced.*

Solution

*Describes the components that make up the general solution, how they relate to one another, and their runtime interactions.*

## Brick and Mortar Language Structure

Brick Patterns

*Patterns for constructing individual (micro)services.*

Mortar Patterns

*Patterns for composing bricks into complete distributed systems.*

## Brick Patterns

Externalization Patterns

*Structural patterns for creating deployable, disposable, and replaceable bricks.*

Externalized Configuration Externalized State Externalized Channels

Runtime Patterns

*Behavioral patterns for creating deployable, replaceable, and observable bricks.*

Runtime Reconfiguration Concurrent Execution Brick Telemetry

## Mortar Patterns

Distributed Systems Patterns

*Composition patterns addressing common distributed systems challenges.*

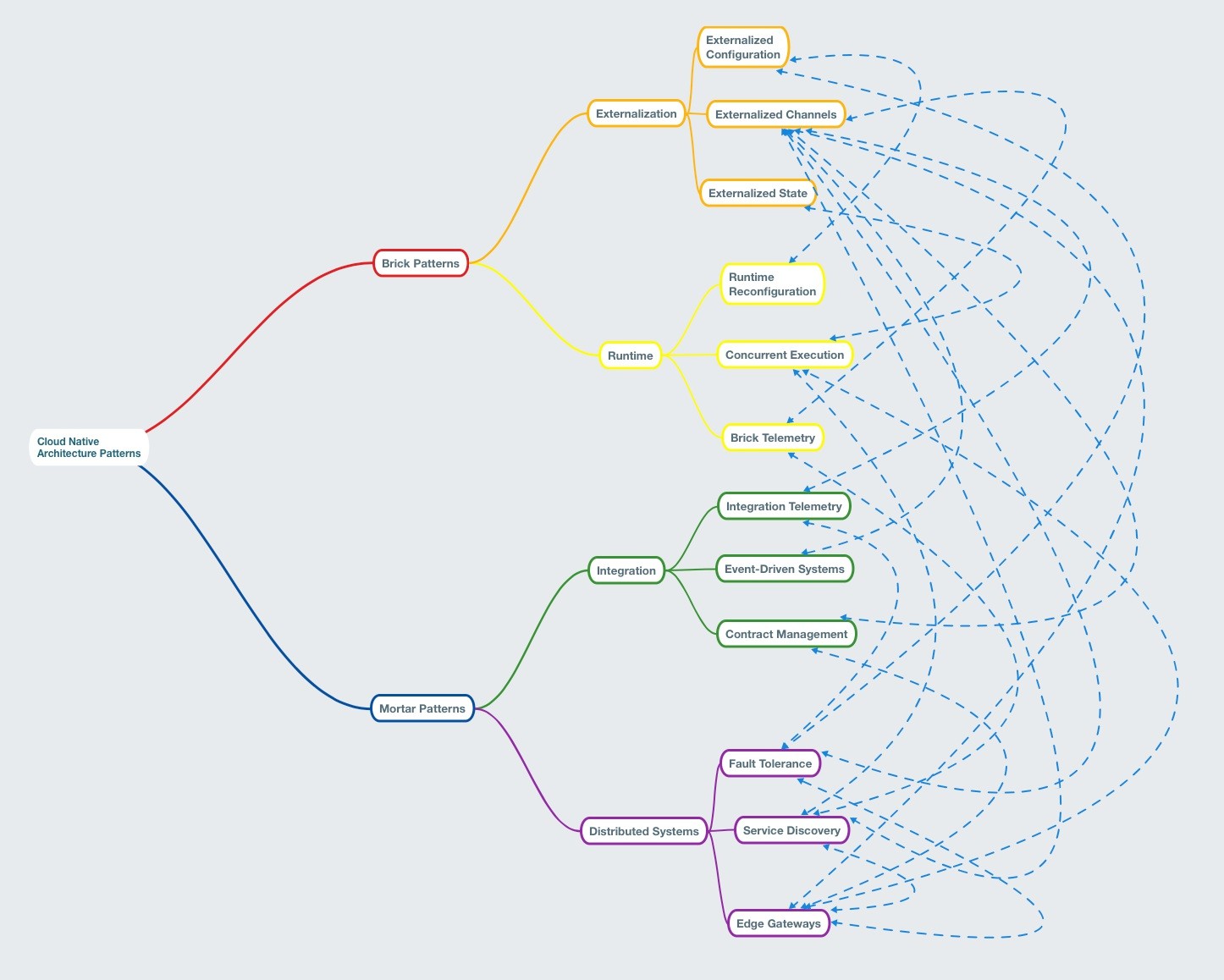
Service Discovery Edge Gateway Fault Tolerance

Integration Patterns

*Composition patterns addressing integration and observability challenges.*

Event-Driven System Contract Management Integration Telemetry

Brick and Mortar Language Relationships



# Brick Patterns

Externalized Configuration Externalized State

Brick Telemetry

# Externalized Configuration

Context

*An application's configuration will vary independently from its code throughout its lifecycle.*

# Problem

*Traditional techniques for managing configuration tightly couple these two orthogonal concepts.*

# Forces

Different environments will have different configuration settings:

resource handles to the database (e.g. a JDBC URL)

credentials to external services (e.g. Amazon S3) per-deploy values such as the canonical hostname (e.g. blog-test.example.com vs. blog- prod.example.com)

features that are toggled on or off

# Forces

Configuration is often bundled within deployment artifacts (e.g. Java properties files).

Build processes often modify configuration based on arguments.

The Deployment Pipeline should only build each deployment artifact once, and deploy the same artifact to multiple environments.

Externalized State

Context

*Disposability and Replaceability require the elimination of "snowflake deployments" from the architecture.*

# Problem

*Traditional state management techniques prevent us from achieving "phoenix deployments."*

# Forces

Early web architectures emphasized server-side state management:

Fat Clients to Thin Clients

Vertically Scaled Cache Management

Stateful Scaffolding on Stateless Protocol (HTTP)

# Forces

Cloud Infrastructure:

Resource Limited Horizontal Scale

Limited Load Balancer Support for Sessions Limited (No) Support for Persistent Local Disk

Brick Telemetry

Context

*Realizing the DevOps Way of Feedback requires that we have visibility into both the business value and technical behavior generated by our services.*

# Problem

*Common approaches to service visibility fall short of the architectural qualities that we need.*

# Forces

Visibility is often accomplished via post facto application of agent-based monitoring tools.

Agent-based monitoring tools don't understand business value.

Determining an application's health often requires complex logic.

Traceability of an application is difficult (or impossible) to accomplish with OTS solutions.

Mortar Patterns

Service Discovery Edge Gateway Fault Tolerance

# Service Discovery

Context

*Decomposition of architecture into services leads to increasingly more distributed systems.*

# Problem

*As systems become distributed, and as service instance lifecycles become more dynamic and independent, location of and communication with dependencies becomes more challenging.*

# Forces

Cloud platforms often assign auto-generated, internal hostnames or private IP's to service instances.

As services are scaled and unhealthy instances are replaced, the addresses of a service's instances are constantly changing.

Binding a service to anything other than logical names for its dependencies leads to friction in the architectural lifecycle.

# Forces

Applying Concurrent Execution is made more difficult (or impossible) when binding services to fixed addresses for their dependencies.

We may want to remove a service instance from the available pool but keep it running to troubleshoot a problem.

Edge Gateway

Context

*Decomposed architectures must always be recomposed. This recomposition often happens within the user interface layer of an application.*

# Problem

*Recomposing an architecture within the User Interface layer presents significant complexities that can lead to decreased agility and degraded user experience.*

# Forces

Systems often must support multiple user experience options (web/mobile/AVR).

Recomposing architectures as the UI layer can require exposing the architecture to the public network.

API needs for a mobile device are often quite different from a web UI.

# Forces

Exposing a network graph to mobile devices can increase latency, increase data usage, and degrade battery life.

UI platforms may not support the integration architecture used for all services.

Native apps often have longer upgrade cycles. Recomposing the architecture there can lead to friction in the architectural lifecycle.

Fault Tolerance

Context

*In order to accomplish its assigned tasks, each brick will need to communicate with other bricks, and with external systems, to which we'll collectively refer as dependencies.*

# Problem

*When a brick's dependencies become unhealthy, unreachable, or slower than normal to respond, that brick's own performance is degraded, and such degredation can potentially cascade across the entire architecture.*

# Forces

The network is not reliable.

Latency is non-zero and unpredictable.

Service availability is a product of its dependencies' availabilities.

# Forces

Failures can be transient. Failures can cascade.

An incorrect or stale response is often preferable to no response.