# Guidelines for Transitioning to Microservices Architecture

## Introduction

This document provides guidance for program managers overseeing the transformation from monolithic applications to microservices. It focuses on key architectural and operational practices, emphasizing the need for an understanding of Agile methodologies, Scrum, DevSecOps, and the Strangler Pattern.

## Microservices

Microservices involve structuring an application as a collection of loosely coupled services. This enhances modularity, simplifies development, testing, and maintenance, and supports parallel development across small, autonomous teams. It also enables continuous delivery and deployment by allowing each service to be developed and scaled independently.

## Strangler Pattern

The Strangler Pattern facilitates the gradual replacement of monolithic architecture components with microservices. This includes setting up a proxy to manage traffic, implementing new services, and gradually taking over functionality from the old system until the monolith can be decommissioned.

## Key Decisions

Key decisions during the transition include the adoption of Infrastructure as Code, choice of programming languages and frameworks, use of separate databases for each service, and implementation of security practices like encryption and secure transport protocols.

## Recommendations

The recommendations provide a comprehensive approach to implementing a robust microservices architecture, including defining bounded contexts, ensuring statelessness, and using Domain-Driven Design. Key practices such as deploying services in containers, using RESTful APIs, and implementing CI/CD pipelines are emphasized to enhance agility and system resilience.

### Detailed Recommendations

* - Define Bounded Context: Understand and define the bounded context for each service.
* - Statelessness: Ensure that services are stateless to improve scalability and resilience.
* - Cattle Not Pets: Treat servers and services like cattle, not pets; do not get attached to individual instances.
* - Separate Data Store: Use a separate database for each microservice to ensure loose coupling and independent scalability.
* - Self-contained Services: Design services to be self-contained, with all necessary components to perform their function.
* - Loosely Coupled: Maintain loose coupling between services to enhance flexibility and maintainability.
* - Idempotent Services: Ensure your services are idempotent.
* - Asynchronous vs. Synchronous: Decide which services should be asynchronous and which should be synchronous.
* - API Gateway vs. Service Mesh: Understand the roles of API Gateway and Service Mesh and define authentication/authorization processes.
* - Use JWT at the Gateway/Mesh layer for centralized authentication.
* - Check Inter-Service Communication: Monitor communication volumes between services to resolve coupling issues.
* - Use REST whenever possible for simplicity and standardization.
* - Deploy microservices in containers to enhance portability and scalability.
* - Implement CI/CD pipelines for each microservice using the same platform but distinct configurations.
* - Leverage DevSecOps practices to ensure security from the start.
* - Use chaos engineering principles to test and improve system robustness.
* - Prefer JSON for data interchange instead of XML.
* - Use YAML for configuration management.
* - Implement caching strategies to reduce load and improve performance.
* - Design each service to manage its own authorization while employing a global authentication strategy.
* - Use monitoring tools like Prometheus to automate health monitoring and issue detection.
* - Implement tracing to monitor requests across services, aiding in debugging and performance optimization.