DoD Enterprise DevSecOps Initiative
Moving to Microservices

v.1.3
DRAFT - UNCLASSIFIED – PRE-DECISIONAL

Table of Contents
Introduction .................................................................................................................................................. 2

Microservices ............................................................................................................................................ 2
Strangler Pattern ....................................................................................................................................... 3

Key Decisions ................................................................................................................................................ 3
Recommendations: ....................................................................................................................................... 4
Read more ..................................................................................................................................................... 5
Introduction

This document is intended to help DoD Program Managers understand the challenges and decisions that have to be made to move from Monolithic applications to Microservices. It is not intended to train readers on what Microservices are. It is understood that the reader must already understand the basics of Agile, Scrum, DevSecOps (DevOps) and Microservices. It is important as well to understand the Strangler Pattern concept.

Microservices

A 'microservice' is a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services.

In a Microservices architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop, test, and more resilient to architecture erosion. It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.

It also allows the architecture of an individual service to emerge through continuous refactoring. Microservices-based architectures enable continuous delivery and deployment.
Strangler Pattern

Martin Fowler describes the Strangler Application:

- One of the natural wonders of this area are the huge strangler vines. They seed in the upper branches of a fig tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host.
- To get there, the following steps were followed:
  - First, add a proxy, which sits between the legacy application and the user. Initially, this proxy doesn’t do anything but pass all traffic, unmodified, to the application.
  - Then, add new service (with its own database(s) and other supporting infrastructure) and link it to the proxy. Implement the first new page in this service. Then allow the proxy to serve traffic to that page (see below)
  - Add more pages, more functionality and potentially more services. Open up the proxy to the new pages and services. Repeat until all required functionality is handled by the new stack.
  - The monolith no longer serves traffic and can be switched off.


Key Decisions

- Infrastructure as Code: all configs, including RBAC, SDN, load balancing etc., should be in code
- Programming language per microservices (couple of options such as Java, Python, Go...). Select one framework per programming language like Spring for example.
- Databases per service
- Encrypted by default (part of Service Mesh usually for west/east traffic)
- Transport (RPC/HTTPs etc.)
- API Gateway (understand difference between internal/external traffic mesh vs gateway) (can be provided by Service Mesh or Kubernetes)
- Service Mesh
- Authentication/Authorization (JSON Web Token (JWT) etc.) (part of Service Mesh or API Gateway usually). Please note that you will also create your own auth microservice for your internal fine grained auth.
- Access Control (ACL) (part of Service Mesh usually)
- Messaging
- Storage management (for persistent storage)
- Health/Readiness (part of Kubernetes/Docker)
- Logs (part of Kubernetes)
- Telemetry (understand the difference with traditional logs)
- Monitoring solution
- Observability (Tracing)
- Circuit breaker (part of Service Mesh usually)
- Use bounded retries and timeouts (part of Service Mesh usually)
- High Availability (part of Kubernetes and/or Service Mesh)
- Load Balancing (part of Kubernetes and/or Service Mesh)
- Service Discovery (part of Kubernetes and/or Service Mesh)
- Canary / Traffic Management (part of Service Mesh usually)
- Backups (storage + containers + kubernetes config)
- Key Management and Certificate management (part of Service Mesh usually)
- Centralized secrets management (usually managed by Kubernetes)

Note: Kubernetes with ISTIO as a Service Mesh can provide a lot of these features by DEFAULT.

See: [https://istio.io/docs/concepts/security/](https://istio.io/docs/concepts/security/) and more at istio.io

**Recommendations:**

- Stateless
- Cattle not pets!
- **Separate data store (database) per microservice**
- Self-contained services
- Loosely coupled
- Ensure your services are idempotent
- Define which service can be asynchronous vs synchronous.
- Understand API Gateway vs Service Mesh and clearly define Authentication/Authorization process between services and User auth.
- Understand JSON Web Tokens to achieve the end goal of creating a distributed authentication mechanism for Microservices [https://nordicapis.com/how-to-control-user-identity-within-microservices/](https://nordicapis.com/how-to-control-user-identity-within-microservices/).
- Understand volume of communications between two services to verify coupling.
- Leverage DDD (Domain driven design) [https://en.wikipedia.org/wiki/Domain-driven_design]
- 2 pizza team (separate team for each microservice)
- Use REST whenever possible
- Use JWT at the Gateway/Mesh layer for centralized auth.
- Separate Shared Libraries from Microservices
- Leverage messaging and asynchronous communication when possible
- Use event-driven architecture to ensure that when something happens to one service, the other services can follow up with their own actions.
- Check your inter service communication volumes, if too high, you have a coupling issue.
- Recommend sticking to one or two programming languages as it can create complexity within teams.
• Leverage Zero Trust model (all blocked by default) and whitelisting. (part of Service Mesh usually)
• Deploy Microservices in containers
• Keep in mind graceful failure is key and avoid single points of failure!
• Recommend using micro segmentation to segment Microservices clusters. (part of Service Mesh usually)
• Use whitelisting for access and NOT blacklisting using fine grained role based access control (RBAC) (part of Service Mesh usually)
• Have a CI/CD pipeline per microservice with separate builds per microservice. You can use the same DevSecOps platform for all services. Eg. Same Jenkins but a Jenkinsfile per microservice.
• Leverage DevSecOps and CI/CD with DoD hardened containers
• Leverage Chaos concepts to try to break things before they actually really break!
• Use JSON instead of XML whenever possible
• Use YAML for configurations
• Use caching when possible.
• Each service should do its authorization but global authentication can be used.
• Use Alerts solutions such as Prometheus to automate health/issue detection
• Leverage Tracing to track requests through multiple services and properly debug issues

Read more

• https://microservices.io/patterns/microservices.html - microservices patterns
• https://dzone.com/articles/top-5-microservices-architecture-and-design-best-p - microservices architecture
• https://medium.freecodecamp.org/follow-these-practical-principles-and-get-well-designed-microservices-boundaries-ef2deff69e3 - how to define Microservices boundaries
• https://medium.com/microservices-in-practice/microservices-in-practice-7a3e85b6624c - microservices in practice
• https://nordicapis.com/how-to-control-user-identity-within-microservices/ - how to control user identity
• https://medium.com/technology-learning/how-we-solved-authentication-and-authorization-in-our-microservice-architecture-994539d1b6e6 - how solve auth
• https://auth0.com/blog/introduction-to-microservices-part-4-dependencies/ - how to share data between services
• http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/ Microservices architecture