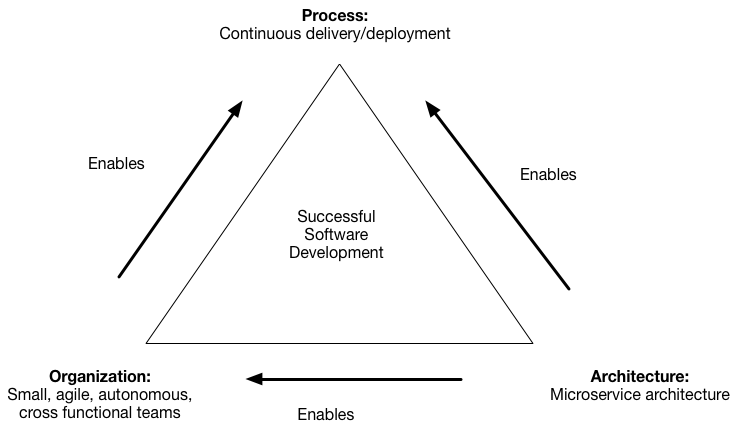
**Microservices Pattern**

**1 Description**

Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams. Small is not defined as a number of lines of code, but rather as a container that concentrates on solving 1 business problem. Micro is meant to signify the amount of business problems we are solving, rather than size.

**A service is a mini-application that implements narrowed focused functionality such as order management, customer management etc.**

A major goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment.



The microservice architecture does this in two ways:

1. Simplifies testing and enables components to deployed independently
2. Structures the engineering organization as a collection of small (6-10 members), autonomous teams, each of which is responsible for one or more services

**Microservices are not**

* A simple API in front of complex monolithic services
* A large, coarse-grained service or monolithic set of services packaged in a Docker container
* A service exposed via API by another party
* Coupled to other assets requiring dependent deployment
* Delivered without automation of testing and deployment
* Built on mutable compute infrastructure that is updated and patched separately from software deployment

**1.1 Pattern Information**

|  |  |
| --- | --- |
| **Tag** | **Description** |
| Pattern Family | Service |
| Pattern ID | Microservice |
| Pattern Status | Draft |
| Version | 0.1 |
| Author(s) | Jamie Gunn |
| Sources | <http://microservices.io/patterns/microservices.html> |

**1.2 Context**

In the development of a server-side enterprise application, you want an architecture that supports the following:

* Desktop browsers
* Mobile browsers
* Other systems
* Native mobile applications.
* Expose an API for 3rd parties to consume.

It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.



**1.3 Problem**

We are striving to improve simplification of testing, deployment, and creating consistent organization around how source code is structured with respect to Business Capability.

Problems that are being addressed include:

* Developer independence & Slow day to day development
* Obstacles to agile development and deployment
* Scaling the application can be challenging
* Reliability of the application
* Long-term commitments to a technology stack
* Lifecycle automation

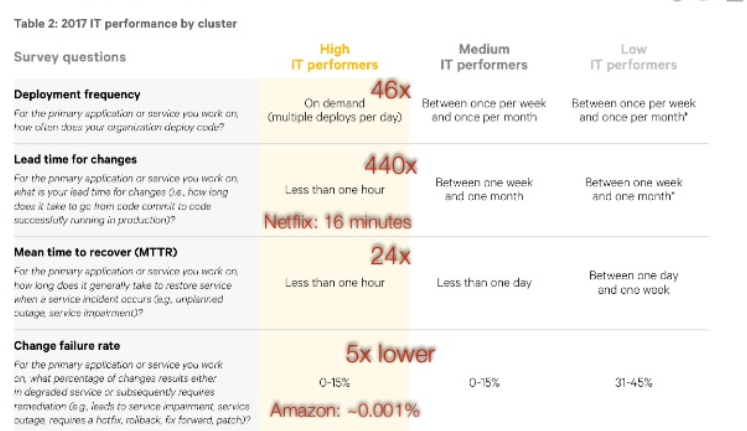
Microservices allow us to attack the “Quality Attributes” of the source code, sometimes people call these the “Non-Functional Requirements”.

Microservice architectures are split along business domain boundaries, which asks us to talk about:

* Domains: customer, product, order, notification, …
* Processes: customer onboarding, order fulfillment, …

**Instead of talking about** SAP, Salesforce, mainframe, …

**IT Performance**: Please see the below metric regarding how well high-performing IT organization perform on some of the “Non-Functional Requirements”.



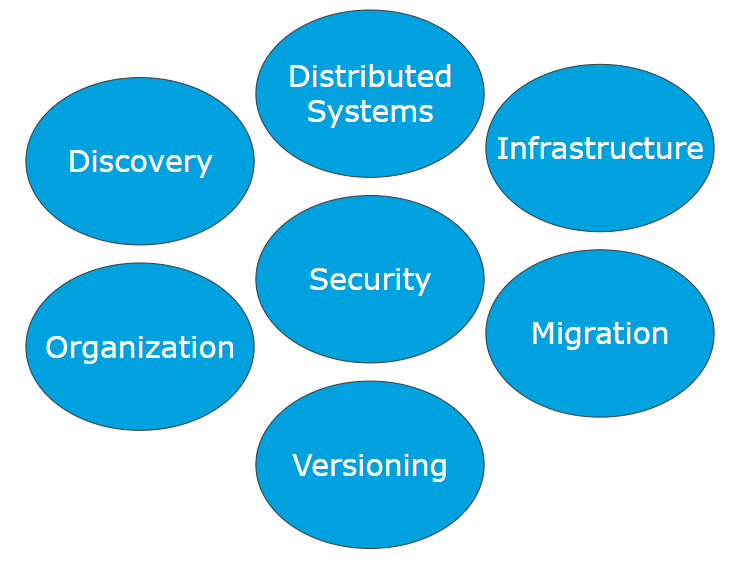
**1.4 Forces**

Forces For:

* **Agility -** Microservices foster an organization of small, independent teams that take ownership of their services.
* **Flexibile Scaling -** Microservices allow each service to be independently scaled to meet demand for the application feature it supports.
* **Easy Deployment -** Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn’t work.
* **Technology Freedom -** Microservices architectures don’t follow a “one size fits all” approach. Teams have the freedom to choose the best tool to solve their specific problems.
* **Resilient -** Service independence increases an application’s resistance to failure.

Forces Against:

* **Design -** Finding the right set of services is challenging. Without care, you will end up building dependent services, and are right back where you started.
* **Complexity -** Distributed systems are complex. Messaging is highly complex. Due to multiple moving parts, operating a microservices model is difficult to operate.
* **Coordination -** Deploying features that span multiple services requires careful coordination. If you do create dependent services for valid business reasons, you have to coordinate the roll-out of the services.
* **Adoption -** Deciding when to adopt the microservice architecture is difficult.
* **Challenges -**

****

**2.0 Solution**

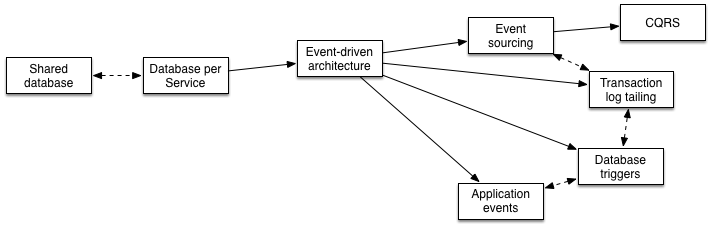
**2.1 Solution Description**

Structure the application as a set of loosely coupled, collaborating services. Each service implements a set of narrowly, related functions. For example, an application might consist of services such as the order management service, the customer management service etc. Please review the following decomposition patterns for proper ways in which to decompose your system into services.

* [Decompose by business capability](http://microservices.io/patterns/decomposition/decompose-by-business-capability.html)
* [Decompose by subdomain](http://microservices.io/patterns/decomposition/decompose-by-subdomain.html)

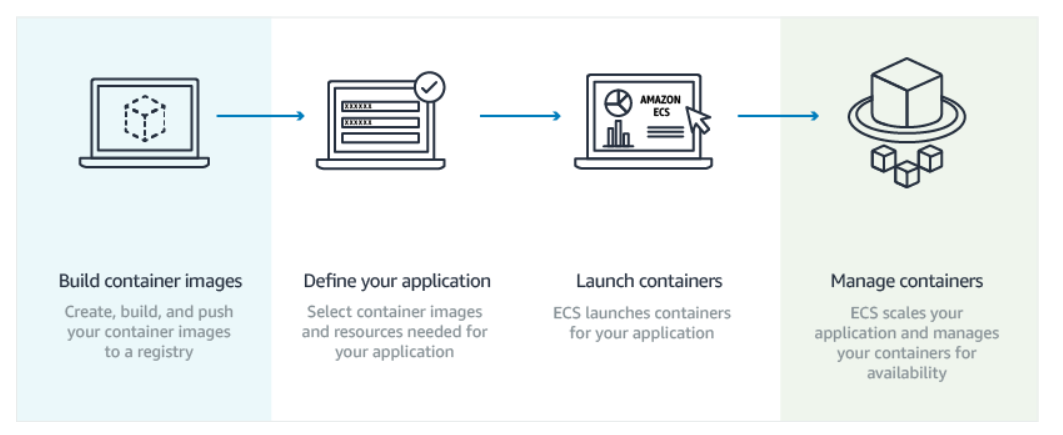
Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. Services can be developed and deployed independently of one another.

Each service has its [own database](http://microservices.io/patterns/data/database-per-service.html) in order to be decoupled from other services. Data consistency between services is maintained using asynchronous messaging.

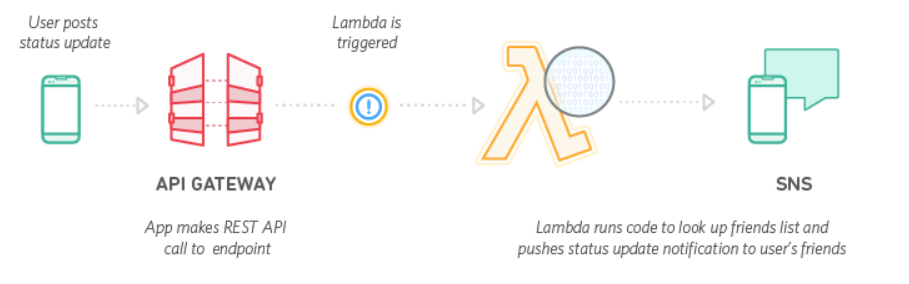


Deployments will be done on either the two following basis:

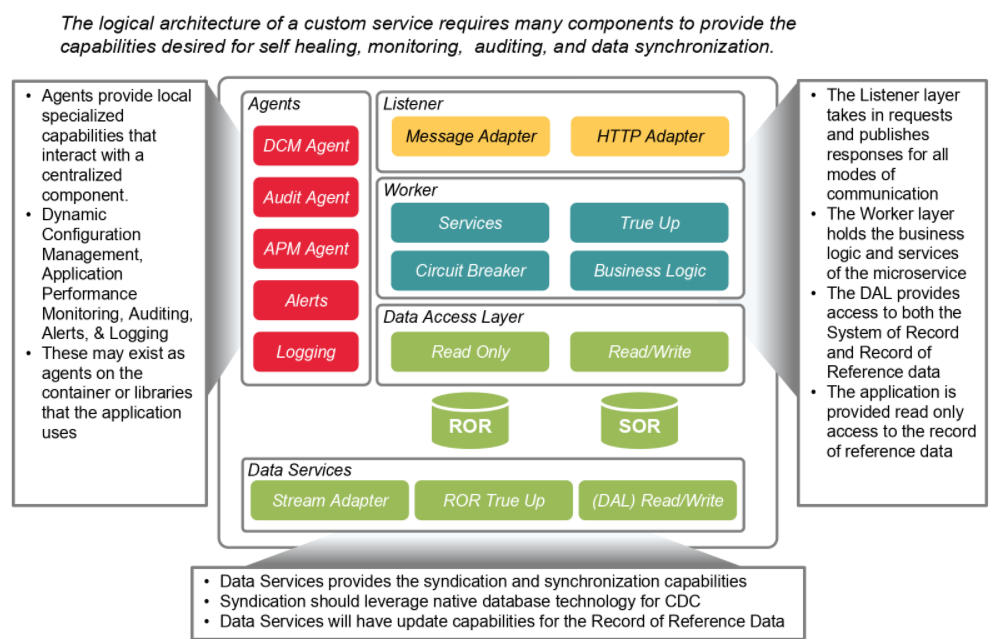
* [Service instance per Container](http://microservices.io/patterns/deployment/service-per-container.html) - <https://aws.amazon.com/ecs/>

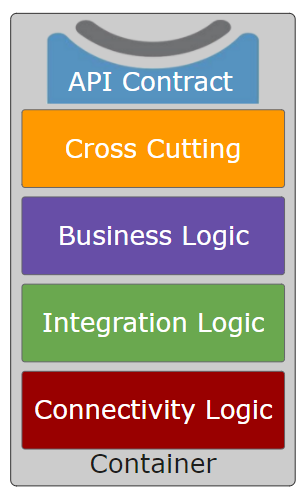


* [Serverless deployment](http://microservices.io/patterns/deployment/serverless-deployment.html) - <https://aws.amazon.com/lambda/>



The following is a proposed reference architecture of how to logically organize a microservice.





**2.2 Applying the Pattern**

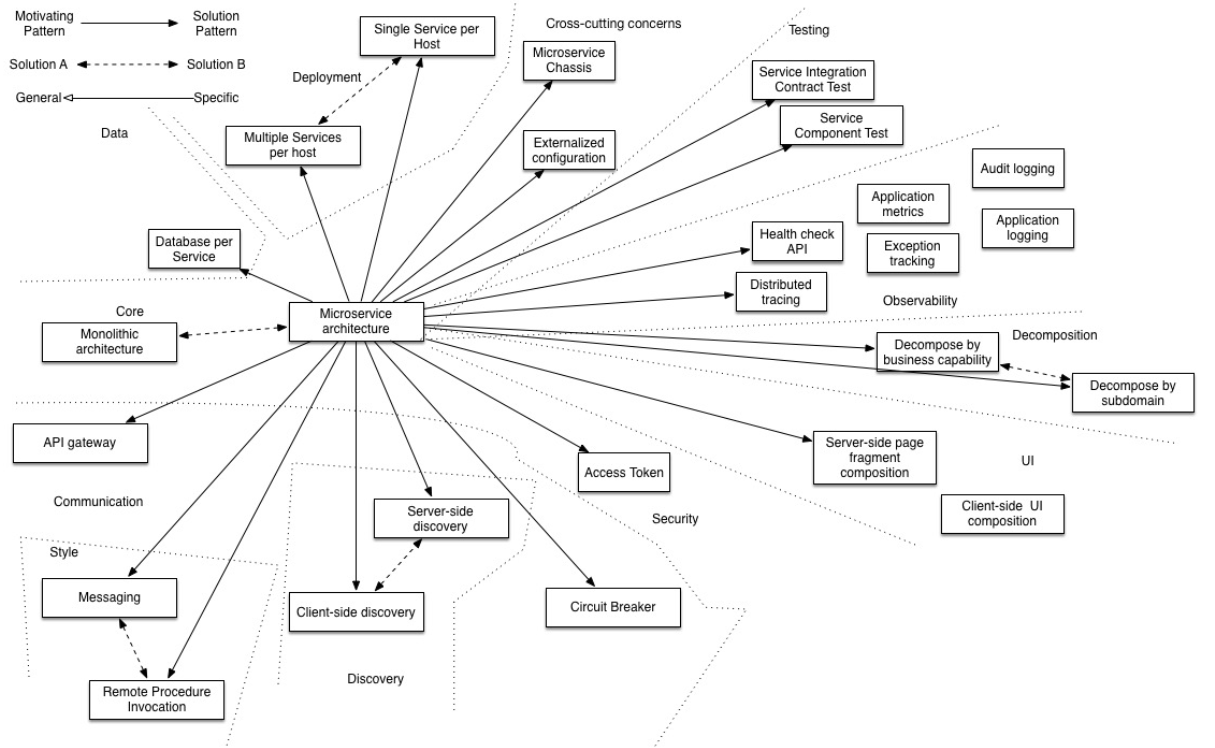
One challenge with using this approach is deciding when it makes sense to use it. Due to its complexity, some applications do not require such in-depth architecture. A challenge that must be overcome include the thought process into partitioning the system into Microservices.

Key items to consider when decomposing the system into services:

* The architecture must be stable
* Services must be cohesive. A service should implement a small set of strongly related functions.
* Services must conform to the Common Closure Principle - things that change together should be packaged together - to ensure that each change affect only one service
* Services must be loosely coupled - each service as an API that encapsulates its implementation. The implementation can be changed without affecting clients
* A service should be testable
* Each service be small enough to be developed by a “two pizza” team, i.e. a team of 6-10 people
* Each team that owns one or more services must be autonomous. A team must be able to develop and deploy their services with minimal collaboration with other teams.

Other key items to take into consideration:

* Developers must deal with the additional complexity of creating a distributed system.
  + Developer tools/IDEs are oriented on building monolithic applications and don’t provide explicit support for developing distributed applications.
  + Testing is more difficult
  + Developers must implement the inter-service communication mechanism.
  + Implementing use cases that span multiple services without using distributed transactions is difficult
  + Implementing use cases that span multiple services requires careful coordination between the teams
* Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
* Increased memory consumption. The microservice architecture replaces N monolithic application instances with N x M services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.
* Data Consistency



**2.6 Benefits**

* Enables the continuous delivery and deployment of large, complex applications.
  + Better testability - services are smaller and faster to test
  + Better deployability - services can be deployed independently
  + It enables you to organize the development effort around multiple, auto teams. It enables you to organize the development effort around multiple teams. Each team is owns and is responsible for one or more single service. Each team can develop, deploy and scale their services independently of all of the other teams.
* Each microservice is relatively small
  + Easier for a developer to understand
  + The IDE is faster making developers more productive
  + The application starts faster, which makes developers more productive, and speeds up deployments
* Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
* Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.

**2.5 Examples**

* Unknown

**3.0 Architectural Guidance**

**3.1 Rationale**

Microservices are a great pattern to when the goal is to build loosely coupled, independent, autonomous, and leverage automation for testing and deployment.

Alternatives include:

* Traditional API development
* Monolithic development
* Shared Database
* SOA

**3.2 Related Patterns**

* Decomposition patterns
  + [Decompose by business capability](http://microservices.io/patterns/decomposition/decompose-by-business-capability.html)
  + [Decompose by subdomain](http://microservices.io/patterns/decomposition/decompose-by-subdomain.html)
* The [Database per Service pattern](http://microservices.io/patterns/data/database-per-service.html) describes how each service has its own database in order to ensure loose coupling.
* The [API Gateway pattern](http://microservices.io/patterns/apigateway.html) defines how clients access the services in a microservice architecture.
* The [Client-side Discovery](http://microservices.io/patterns/client-side-discovery.html) and [Server-side Discovery](http://microservices.io/patterns/server-side-discovery.html) patterns are used to route requests for a client to an available service instance in a microservice architecture.
* The Messaging and Remote Procedure Invocation patterns are two different ways that services can communicate.
* The [Single Service per Host](http://microservices.io/patterns/deployment/single-service-per-host.html) and [Multiple Services per Host](http://microservices.io/patterns/deployment/multiple-services-per-host.html) patterns are two different deployment strategies.
* Cross-cutting concerns patterns: [Microservice chassis pattern](http://microservices.io/patterns/microservice-chassis.html) and [Externalized configuration](http://microservices.io/patterns/externalized-configuration.html)
* Testing patterns: [Service Component Test](http://microservices.io/patterns/testing/service-component-test.html) and [Service Integration Contract Test](http://microservices.io/patterns/testing/service-integration-contract-test.html)
* [Circuit Breaker](http://microservices.io/patterns/reliability/circuit-breaker.html)
* [Access Token](http://microservices.io/patterns/security/access-token.html)
* Observability patterns:
  + [Log aggregation](http://microservices.io/patterns/observability/application-logging.html)
  + [Application metrics](http://microservices.io/patterns/observability/application-metrics.html)
  + [Audit logging](http://microservices.io/patterns/observability/audit-logging.html)
  + [Distributed tracing](http://microservices.io/patterns/observability/distributed-tracing.html)
  + [Exception tracking](http://microservices.io/patterns/observability/exception-tracking.html)
  + [Health check API](http://microservices.io/patterns/observability/health-check-api.html)
  + [Log deployments and changes](http://microservices.io/patterns/observability/log-deployments-and-changes.html)
* UI patterns:
  + [Server-side page fragment composition](http://microservices.io/patterns/ui/server-side-page-fragment-composition.html)
  + [Client-side UI composition](http://microservices.io/patterns/ui/client-side-ui-composition.html)

**3.3 Known Uses**

* Need more research.