



**Morris & Opazo**  
Business Solutions

- [www.morrisopazo.com](http://www.morrisopazo.com) -

## Building Microservices with the 12 Factor App Pattern



Silver  
**Microsoft Partner**

## Context

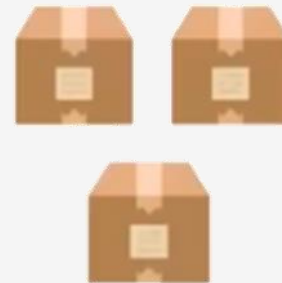
This documentation will help introduce Developers to implementing **MICROSERVICES** by applying the **TWELVE-FACTOR PRINCIPLES**, a set of best practices and methodology for a well-formed architecture, enforcing **AGILE** concepts and favoring **SCALABILITY**

12 Factor App  
Principles



+

Microservice  
Principles



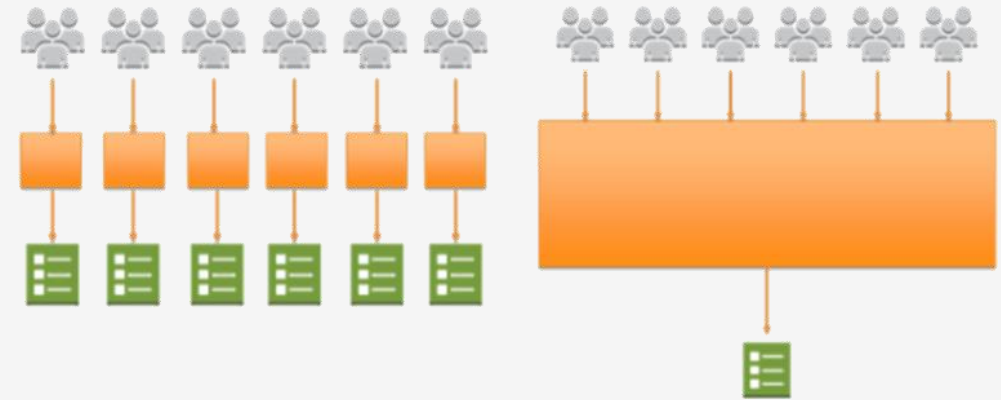
=

Great, Scalable  
Architecture



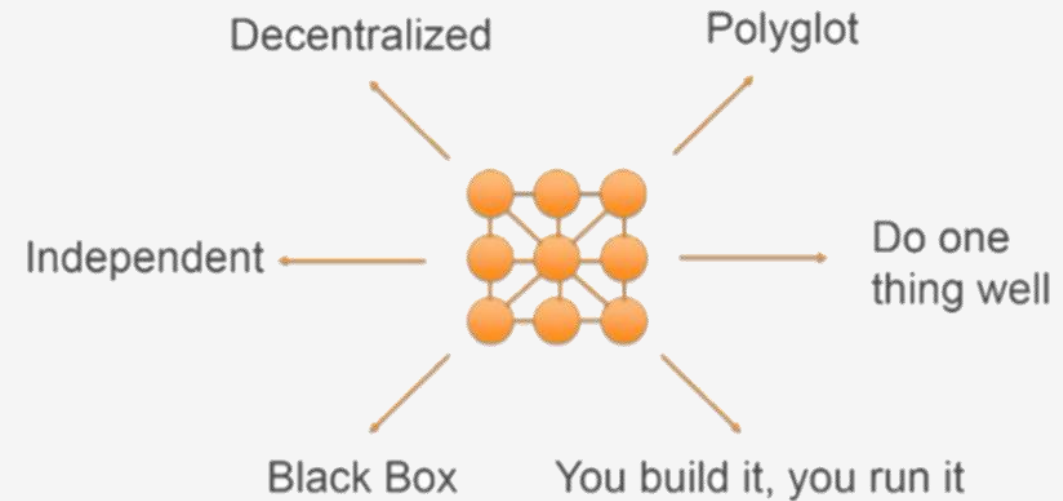
## Benefits of Microservices

- ✓ **AGILITY**, small independent teams take ownership of their services, work independently and quickly (shortening cycle times).
- ✓ **INNOVATION**, small teams can act autonomously and choose the appropriate technologies, frameworks, and tools for their domains.
- ✓ **QUALITY**, improved reusability, composability, and maintainability of code.
- ✓ **SCALABILITY**, Properly decoupled services can be scaled horizontally and independently from each other. The scaling process can be completely automated.
- ✓ **AVAILABILITY**, easier to implement failure isolation, reduce the blast radius of a failing component and improve the overall availability of a given application.



## Principles of Microservices

- ✓ **DECENTRALIZED**, Distributed systems with decentralized data management, development, deployment, and operation. Each microservice has its own view on data models.
- ✓ **INDEPENDENT**, Different components can be changed, upgraded, or replaced independently without affecting the functioning of other components. Teams are enabled to act independently from each other.
- ✓ **DO ONE THING WELL**, Each component is designed for a set of capabilities and focuses on a specific domain.
- ✓ **POLYGLOT PERSISTENCE AND PROGRAMMING**, Heterogeneous approach to operating systems, programming languages, data stores, and tools.
- ✓ **BLACK BOX**, Individual components hide the details of their complexity from other components.
- ✓ **YOU BUILD IT, YOU RUN IT**, The team responsible for building a service is also responsible for operating and maintaining it in production.



## The Twelve Factors

These factors serve as an excellent introduction to the discipline of building and deploying applications in the cloud and preparing teams for the rigor necessary to build a production pipeline around elastically scaling applications.

This methodology helps to build software-as-a-service applications.



[www.morrisopazo.com](http://www.morrisopazo.com) / [contacto@morrisopazo.com](mailto:contacto@morrisopazo.com)  
Chicago - Antofagasta - Temuco - Santiago



Silver  
**Microsoft Partner**

## The Twelve Factors



1) Codebase



2) Dependencies



3) Config



4) Backing services



5) Build, release, run



6) Stateless Processes



7) Port binding



8) Concurrency



9) Disposability



10) Dev/prod parity



11) Logs

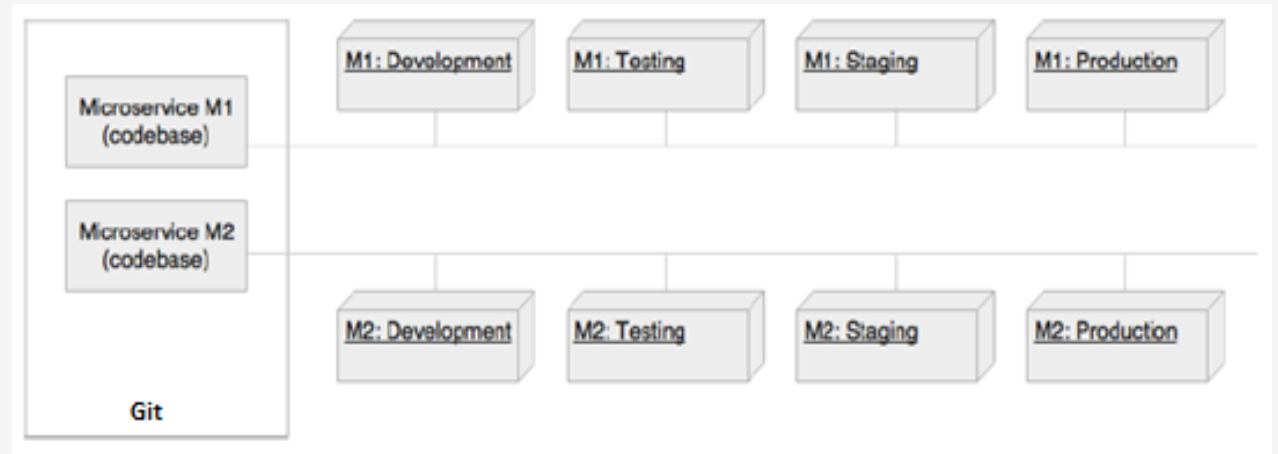


12) Admin processes

## 1) Codebase

*“One codebase tracked in revision control, many deploys”*

- ✓ One Codebase, Multiple Deploys
- ✓ **ANTI-PATTERN**, There must be a change to the codebase to deploy to a specific environment.
- ✓ **ANTI-PATTERN**, Multiple apps sharing the same code.  
**SOLUTION** = Factor shared code into libraries which can be included through a **Dependency Manager**.
- ✓ Code is managed in a distributed source control system such as Git



- ✓ One Codebase = One App
- ✓ Codebase = repo
- ✓ One repo => many deploys
- ✓ App != Many Repos
- ✓ Many Repos = Distributed System

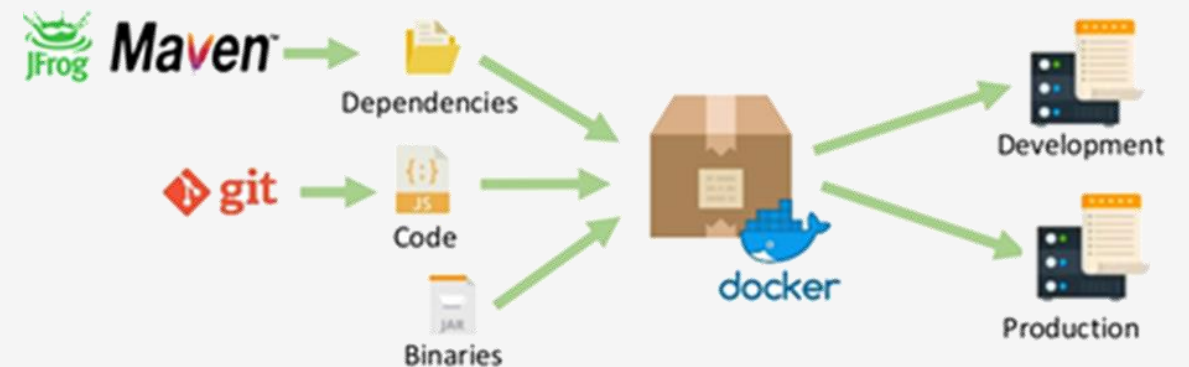
## 2) Dependencies

*“Explicitly declare and isolate dependencies”*

- ✓ **DEPENDENCY MANAGER** as Maven, we explicitly manage dependencies in a pom.xml
- ✓ **DEPENDENCY DECLARATION**, Specify all dependencies via a Dependency Declaration Manifest. Specific versions are important
- ✓ **CENTRAL BUILD ARTIFACT REPOSITORY** such as Jfrog Artifactory, this ensures that the versions are managed correctly
- ✓ **DEPENDENCY ISOLATION**, **Never depend on the host to have your dependency**. Application deployments should carry all their dependencies with them.

### DEPENDENCY DECLARATION (pom.xml)

```
1. ...  
2. ...  
3. <dependencies>  
4. <dependency>  
5. <groupId>group-a</groupId>  
6. <artifactId>artifact-a</artifactId>  
7. <version>1.0</version>
```

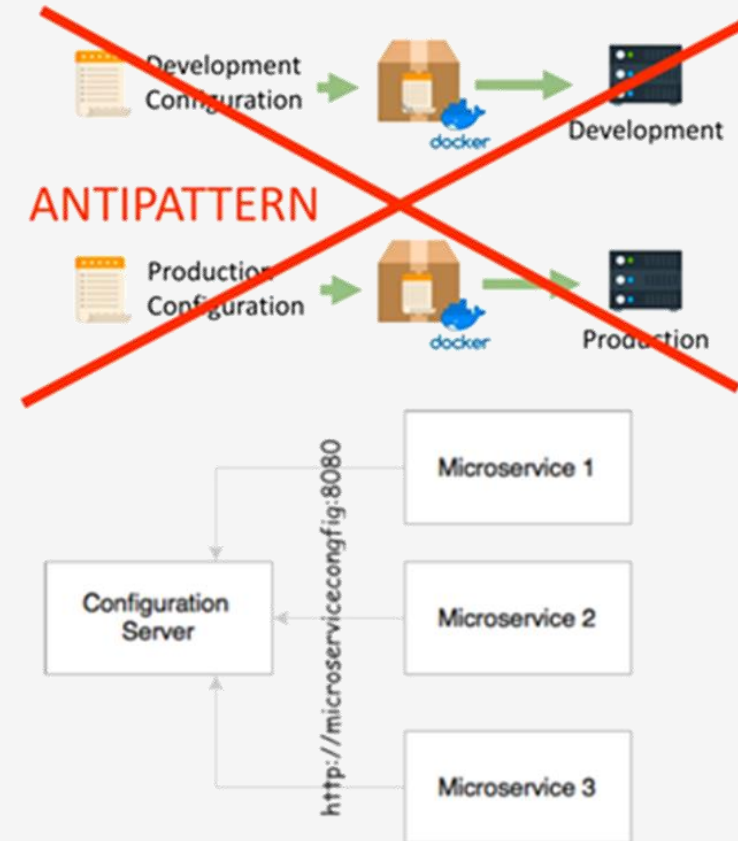




### 3) Config

*"Store config in the environment"*

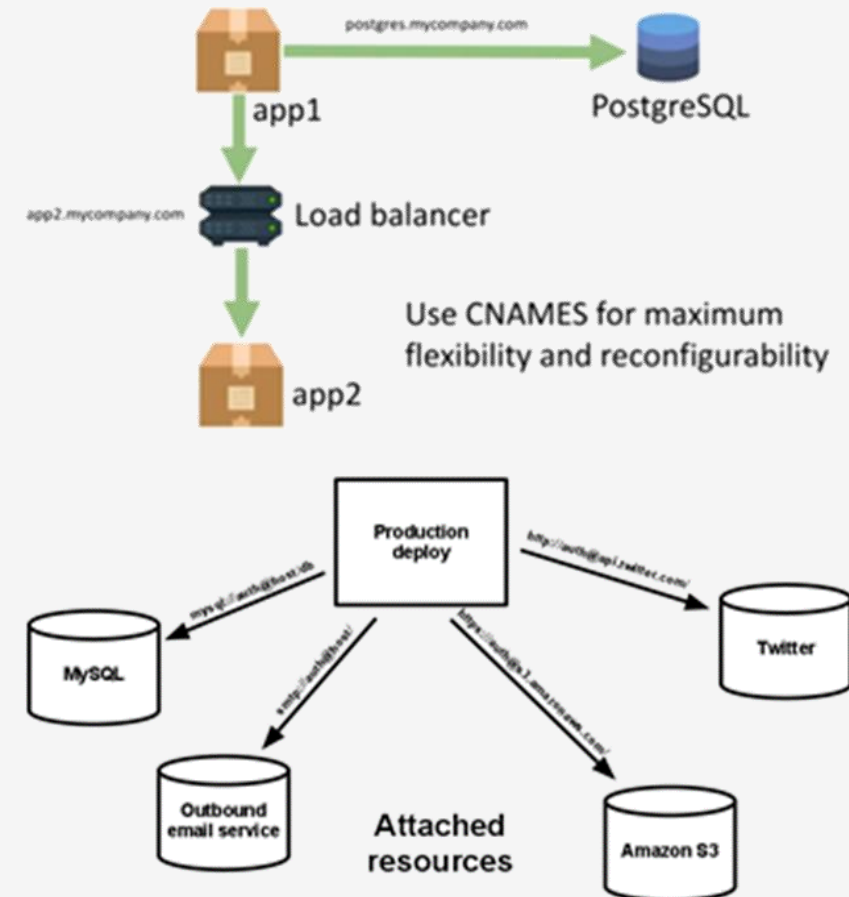
- ✓ Externalization of all configuration parameters from the code. No config in git.
- ✓ An application's configuration parameters vary between environments.
- ✓ Microservices configuration parameters should be loaded from an external source
- ✓ Protect sensitive configuration information (encrypt config settings).
- ✓ Application configuration data is read during service bootstrapping phase.
- ✓ Codebase could be made open source at any moment, without compromising any credentials.
- ✓ Use environment vars



## 4) Backing Services

*"Treat backing services as attached resources"*

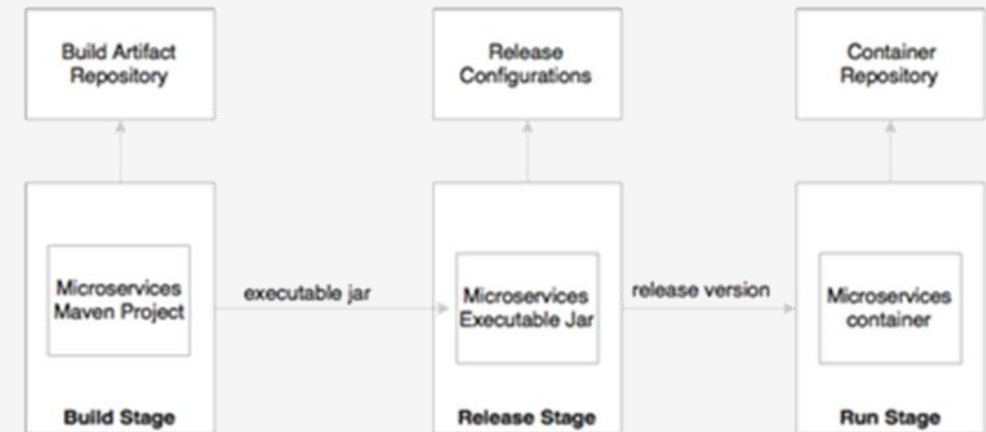
- ✓ All backing services should be accessible through an addressable URL, without complex communication requirements.
- ✓ Make no distinction between local and third party services.
- ✓ Keep Environment Consistence.
- ✓ Examples:
  - Datastores
  - Messaging/Queueing Systems
  - SMTP services
  - Caching system
  - Third-Party APIs.



## 5) Build, Release, Run

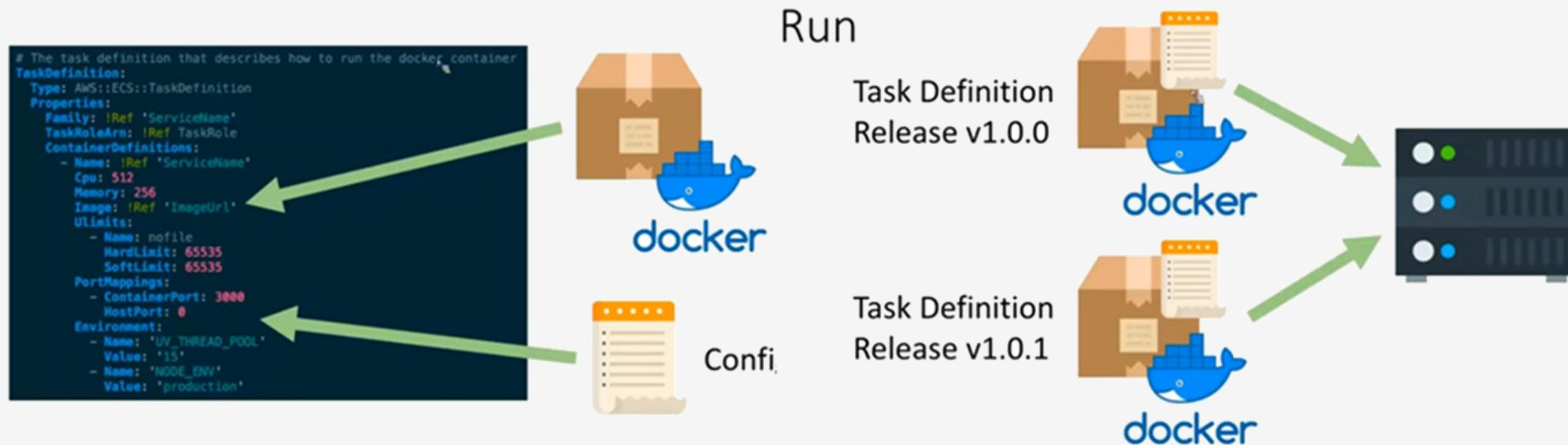
*"Strictly separate build and run stages"*

- ✓ Strong isolation between Build, Release, and Run:
  - **Build Stage**, compiling and producing binaries by including all the assets required.
  - **Release Stage**, combining binaries with environment- specific configuration parameters.
  - **Run Stage**, running application on a specific execution environment.
- ✓ The pipeline is unidirectional, so it is not possible to propagate changes from the run stages back to the build stage.
- ✓ **ANTI-PATTERN**, Specific builds for production.  
**SUGGESTION** = Go through the pipeline.
- ✓ **ANTI-PATTERN**, Make changes to the code at runtime.  
**SUGGESTION** = Any change (or set of changes) must create a new release, following the Pipeline: Build -> Release -> Run.
- ✓ **SUGGESTION** = Every release should always have a unique release ID, such as a timestamp of the release (such as 2011-04-06-20:32:17) or an incrementing number (such as v100).



- ✓ **BUILD** = codebase + dependencies + assets
- ✓ **RELEASE** = **BUILD** + config
- ✓ **RUN** = run process against **RELEASE**
- ✓ **ROLLBACK** = just use the last release instead.

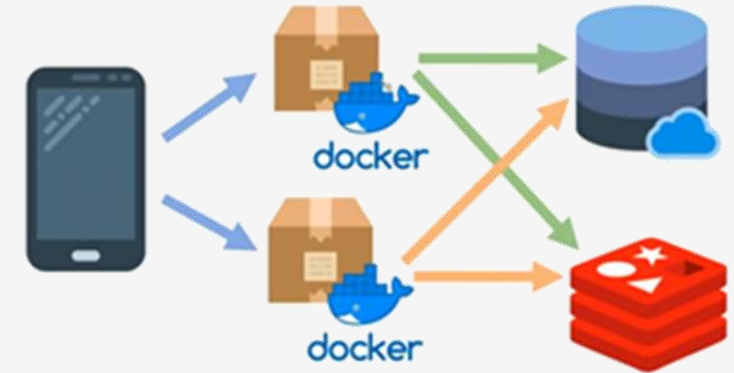
## 5) Build, Release, Run



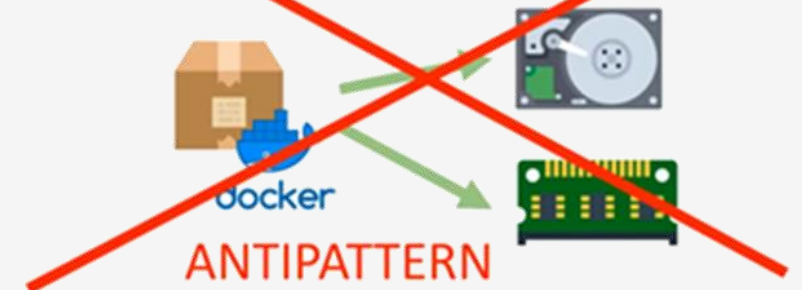
## 6) Stateless Processes

*"Execute the app as one or more stateless processes"*

- ✓ **SUGGESTION**, Processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service.
- ✓ **ANTI-PATTERN**, To assume that anything cached in memory or on disk will be available on a future request or job.
- ✓ **ANTI-PATTERN**, "sticky sessions".  
**SUGGESTION**, Session state data (a datastore that offers time-expiration, such as Memcached or Redis).
- ✓ They can be killed and replaced at any time without the fear that a loss-of-a-service instance will result in data loss.
- ✓ Microservices should always be stateless.



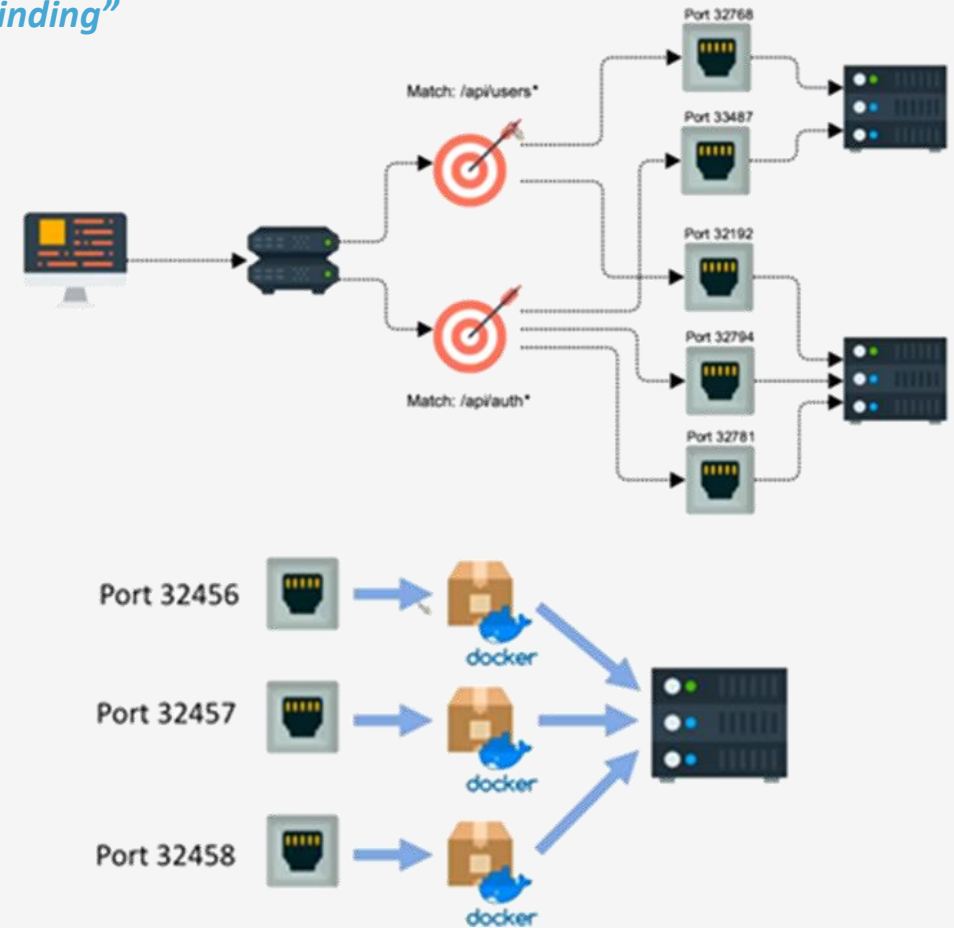
Stateful container stores state in local disk or local memory.  
Workload ends up tied to a specific host that has state data.



## 7) Port binding

*"Export services via port binding"*

- ✓ Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained.
- ✓ Microservices embed service listeners as a part of the service itself.
- ✓ You should run the service without the need for a separated web or application server.





## 8) Concurrency

*"Scale out via the process model"*

- ✓ When you need to scale, launch more microservice instances:
  - Microservices should be designed to scale out by replicating.
  - Microservices should be designed to scale horizontally rather than vertically.

### ✓ AUTO-SCALING

The services can be elastically scaled or shrunk based on given metric.

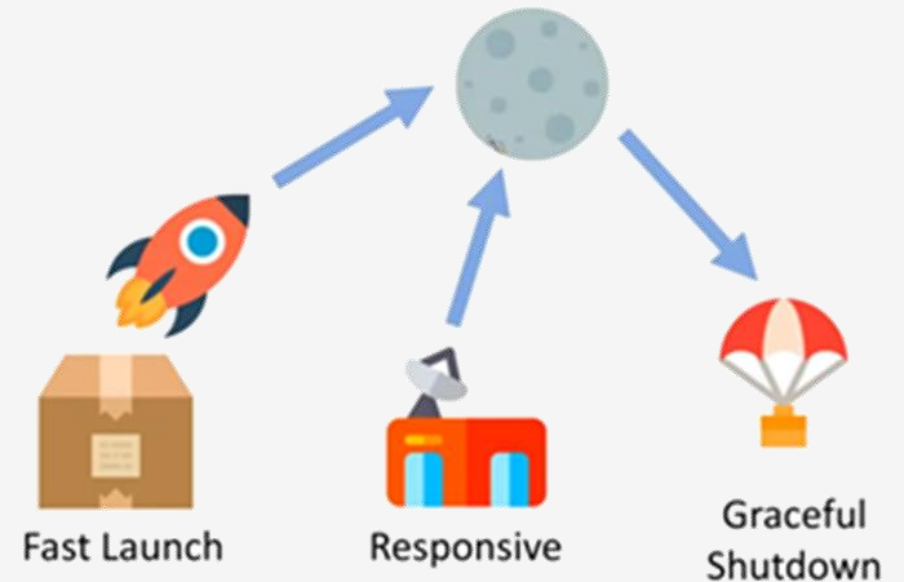
- ✓ Threading can be used within microservices, but don't rely on it as the sole mechanism for scaling.



## 9) Disposability

*“Maximize robustness with fast startup and graceful shutdown”*

- ✓ **Microservices are disposable**, can be started or stopped at any moment.
- ✓ Startup time should be minimized and microservices should shut down gracefully when they receive a kill signal.
- ✓ In an automated deployment environment, we should be able bring up or bring down microservice instances as quick as possible.
- ✓ It is extremely important to keep the size of the application as thin as possible, with minimal startup and shutdown time.
- ✓ Be robust against sudden death. Replace crashed processes faster.





## 10) Dev/Prod Parity

*“Keep development, staging, and production as similar as possible”*

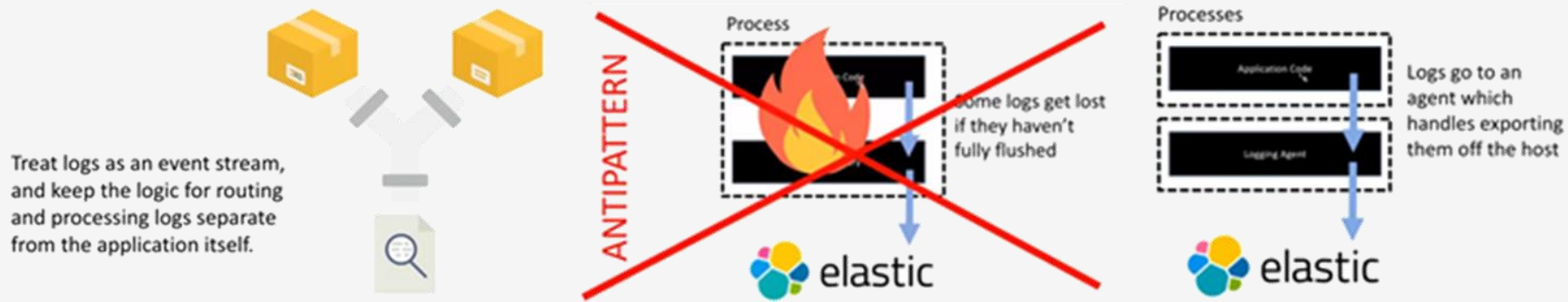
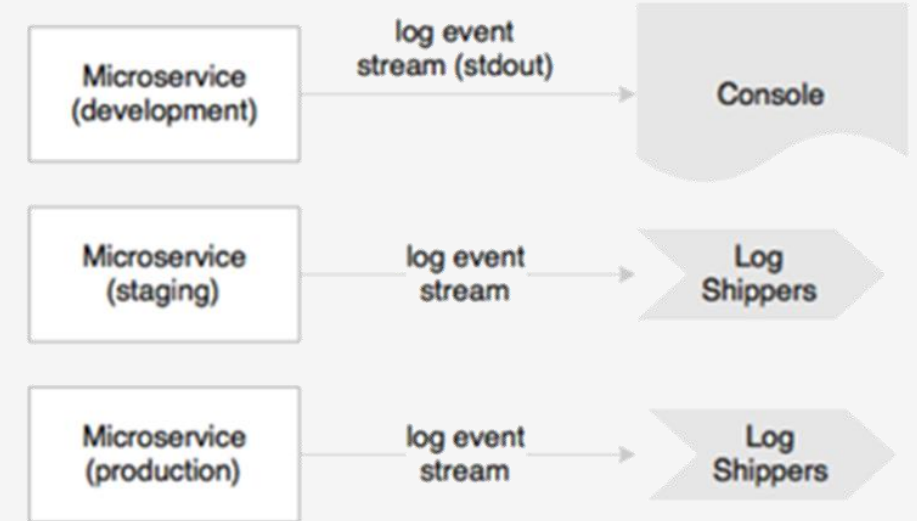
- ✓ The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.
- ✓ Minimize the gaps that exist between all of the environments in which the service runs.
- ✓ As soon as code is committed, it should be tested and then promoted as quickly as possible from Dev all the way to Prod.
- ✓ **ANTI-PATTERN**, In a development environment, run all microservices on a single machine, whereas in production independent machines run each one. If production fails, there is no identical environment to reproduce and fix the issues.

	Traditional app	Twelve-factor app
Time between deploys	Weeks	Hours
Code authors vs code deployers	Different people	Same people
Dev vs production environments	Divergent	As similar as possible

## 11) Logs

*"Treat logs as event streams"*

- ✓ Logs are a stream of events.
- ✓ **ANTI-PATTERN**, Attempt to write to or manage log files.  
**SUGGESTION**, Ship logs to a central repository by tapping the logback appenders and write to one of the shippers' endpoints.
- ✓ Log correlation: All service log entries have a correlation ID that ties the log entry to a single transaction



## 12) Admin Processes

*“Run admin/management tasks as one-off processes”*

- ✓ Use the same release bundle as well as an identical environment for both application services and admin tasks.
- ✓ Admin code should be packaged along with the application code.
- ✓ Admin tasks should never be ad hoc and instead should be done via scripts that are managed and maintained through the source code repository.
- ✓ Admin scripts should be repeatable and non-changing across each environment they're run against.



# Inter-Process Communication in a Microservices Architecture



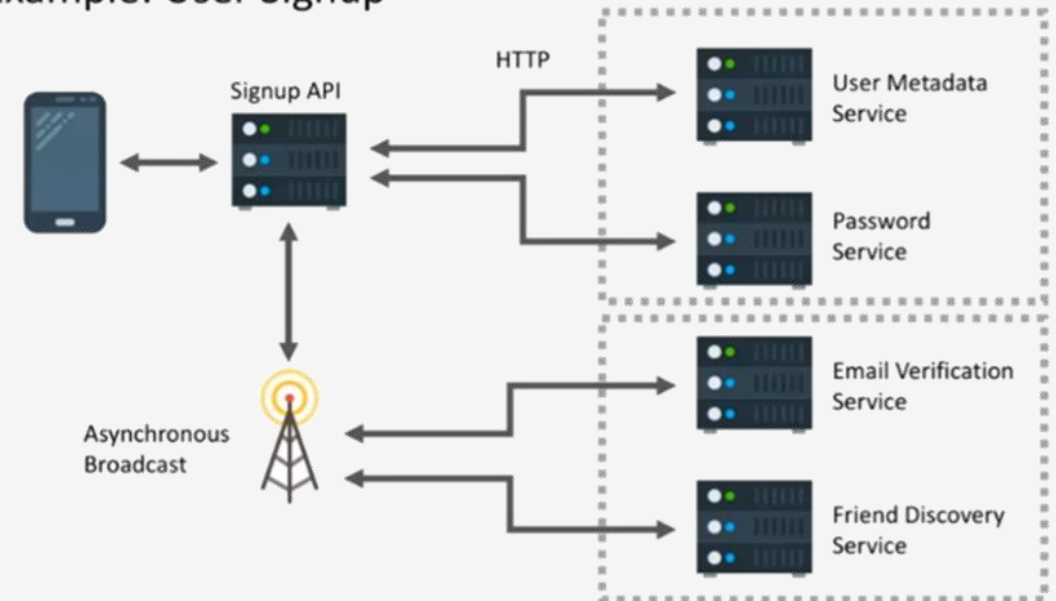
[www.morrisopazo.com](http://www.morrisopazo.com) / [contacto@morrisopazo.com](mailto:contacto@morrisopazo.com)  
Chicago - Antofagasta - Temuco - Santiago



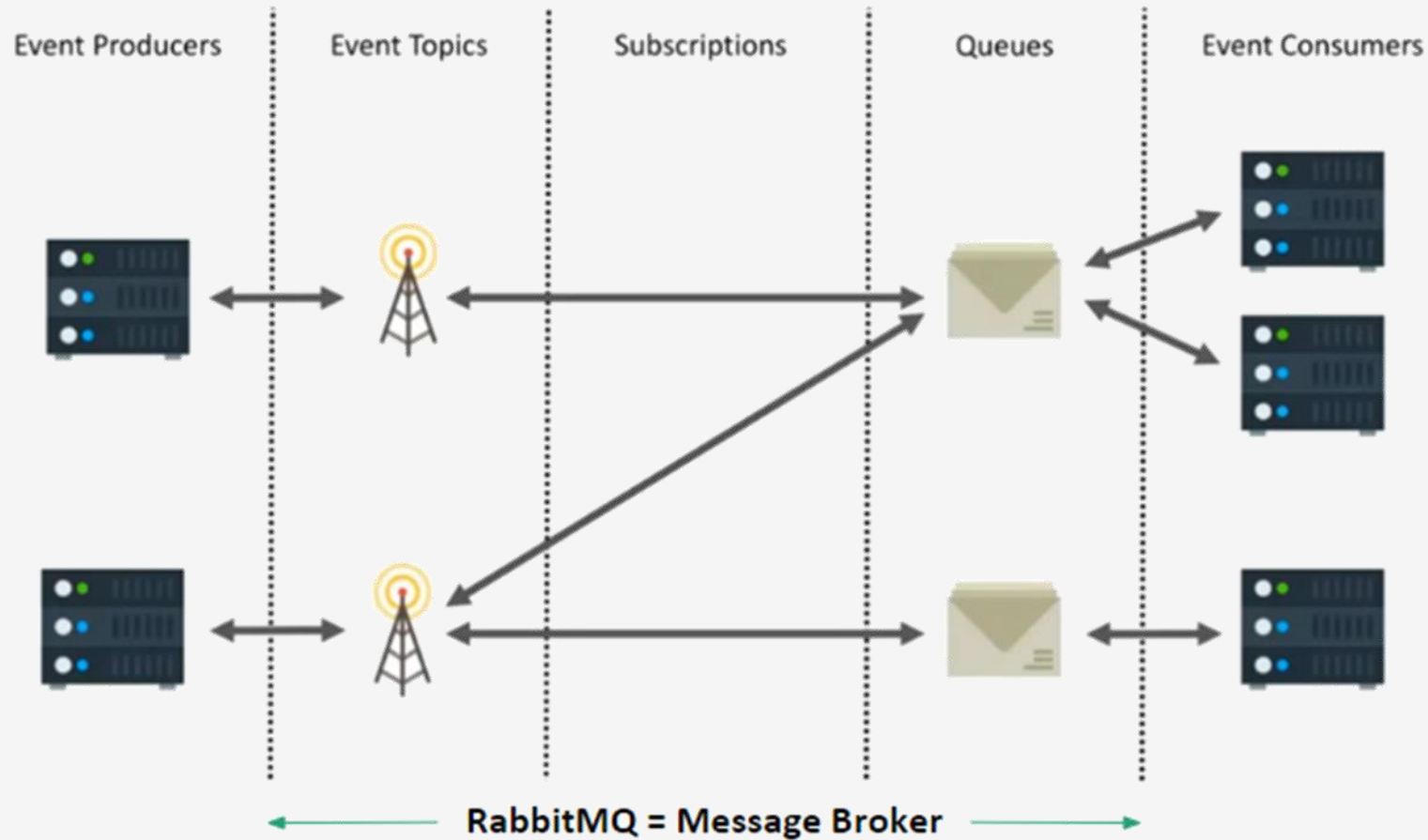
Silver  
**Microsoft Partner**

	One-to-One	One-to-Many
<b>Synchronous</b>	Request/response	—
<b>Asynchronous</b>	Notification	Publish/subscribe
	Request/async response	Publish/async responses

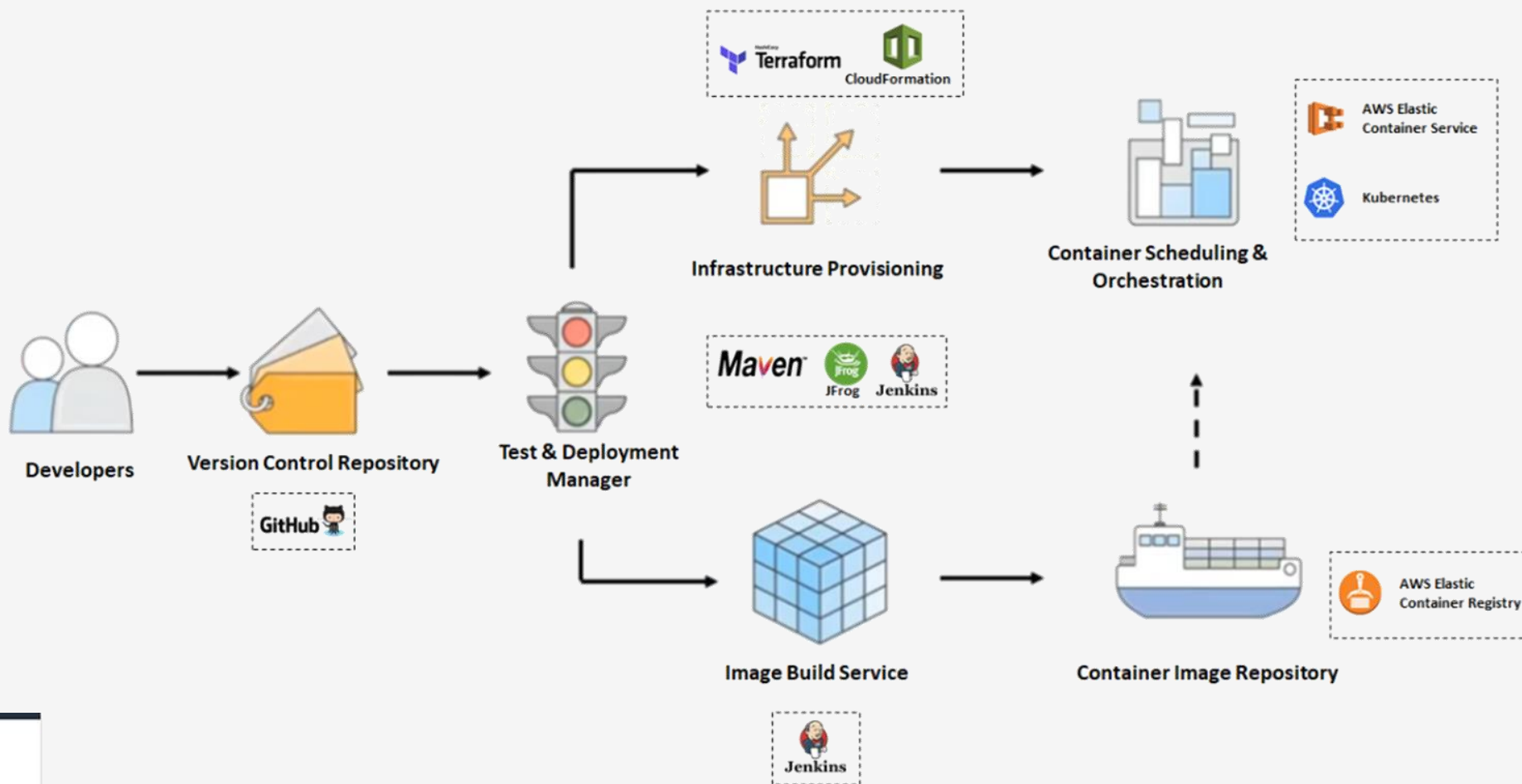
### Example: User Signup



## Asynchronous Microservices Communication through Events



## Microservices Infrastructure Automation




## References

- ✓ The Twelve-Factor App: <https://12factor.net/>
- ✓ Microservices on AWS:  
<https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/microservices-on-aws.pdf>
- ✓ Beyond the Twelve-Factor App (Kevin Hoffman)
- ✓ Spring Microservices (Rajesh RV)
- ✓ Microservice Architecture (Irakli Nadareishvili, Ronnie Mitra, Matt McLarty & Mike Amundsen)
- ✓ Kubernetes Microservices with Docker (Deepak Vohra)
- ✓ Spring Microservices in Action (John Carnell)
- ✓ Spring Boot Messaging (Felipe Gutierrez)







Find out more at  
- [www.morrisopazo.com](http://www.morrisopazo.com)   
*¡Visit our Web Site!*



Silver  
**Microsoft Partner**

**[contacto@morrisopazo.com](mailto:contacto@morrisopazo.com)**  
USA - CHILE

