12-factor apps in .NET

Building apps for the cloud

Ian Cooper @ICooper ian@huddle.com

Who are you?

- Software Developer for 20 years
  - Worked mainly for ISVs
    - *Reuters, SunGard, Misys, Huddle*
  - Worked for a couple of MIS departments
    - *DTI, Beazley*
- Microsoft MVP for C#
  - Interested in OO, SOA, EDA,, Messaging, REST
  - Interested in Agile methodologies and practices
- No smart guys
  - Just the guys in this room

Intelligent collaboration for the Enterprise

BRIGHTER

HOME INTRO TUTORIALS CONCEPTS DOCS

Welcome to Brighter

This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.

It can be used for implementing Ports and Adapters and CQRS (PDF) architectural styles in .NET.

It can also be used in microservices architectures for decoupled communication between the services

GET STARTED

Agenda

Context
Origins
Goals
The 12 Factors
Design
Build and Release
Manage

Housekeeping

Raise your hand if you can't read the codes of the slides

For simplicity, you can download the slides from
https://github.com/iancooper/Presentations

And you can download the code from
https://github.com/iancooper/12FactorTutorial

Which should help from the back of the room.

Context

Why should you care?

12-Factor Apps are one of the pillars of Cloud Native

What is Cloud Native?

Perhaps we first need to ask: what do we mean by cloud?

"By cloud, we mean any computing environment in which **computing**, **networking**, and **storage** resources can be provisioned and released **elastically** in an **on-demand**, **self-service** manner."

Migrating to Cloud-Native Application Architectures, Matt Stine

By Cloud Native we mean:

1. An application that has been designed to run in such an environment

2. An application that allows such an environment to be exploited.

Cloud Native Maturity Model

1. Cloud Ready	• No permanent disk access • Self-contained application • Platform managed ports and networking
2. Cloud Friendly	• 12 Factor App Methodology • Horizontally scalable • Leverages platform for high availability
3. Cloud Resilient	• Fault Tolerant and resilient design • Cloud-agnostic runtime implementation • Bundled metrics and monitoring • Proactive failure testing
4. Cloud Native	• Microservices Architecture • API-first design

Faiz Parker, Pivotal

Increasingly we are seeing the guidelines applied not just to applications that run in the cloud but also those that run in containers

Containers run 'sandboxed executables' with the intent of maximizing utilization of resources to lower costs

Using tools like Swarm, Kubernetes, ACS etc. containers can be scheduled in an elastic, on demand, self-service manner

So what is good for cloud in 12-Factor, is good for containers

Origins

Credit where it is due

The Twelve Factor App

Created by contributors to the Heroku platform (particularly the co-founder Adam Wiggins):

Heroku is a Polyglot PAAS platform

The team derived the guidelines from their experience of what made app successful on Heroku.

The Twelve-Factor App

Introduction

In the modern era, software is commonly delivered as a service: called *web apps*, or *software-as-a-service*. The twelve-factor app is a methodology for building software-as-a-service apps that:

- Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
- Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
- **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
- And can **scale up** without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

Background

The contributors to this document have been directly involved in the development and deployment of hundreds of apps, and indirectly witnessed the development, operation, and scaling of hundreds of thousands of apps via our work on the Heroku platform.

Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project

Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments

Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration

Minimize divergence between development and production, enabling **continuous deployment** for maximum agility

And can **scale up** without significant changes to tooling, architecture, or development practices.

The 12 Factors

The factors that impact how we design applications for the cloud

1. One codebase tracked in revision control, many deploys
2. Explicitly declare and isolate dependencies
3. Store config in the environment
4. Treat backing services as attached resources
5. Strictly separate build and run stages
6. Execute the app as one or more stateless processes
7. Export services via port binding
8. Scale out via the process model
9. Maximize robustness with fast startup and graceful shutdown
10. Keep development, staging, and production as similar as possible
11. Treat logs as event streams
12. Run admin/management tasks as one-off processes

Design

The factors that impact how we design applications for the cloud

Port Binding

“Export services via port binding”

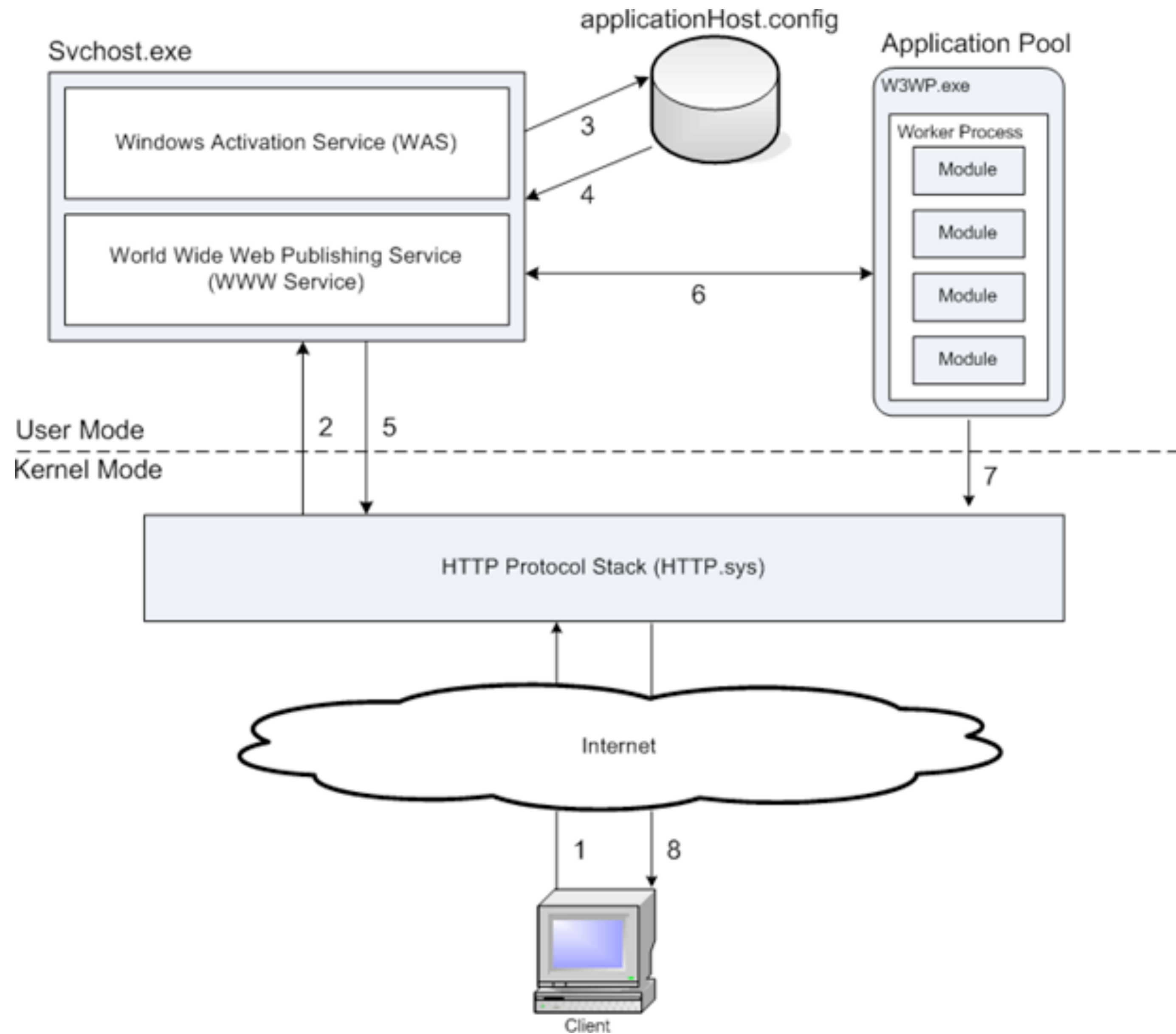
Your app should bind to a port, and receive requests via that port, or return responses

A web app tends to listen to HTTP, but a backing service might listen to AMQP where the ’port’ is a URL to the server, etc.

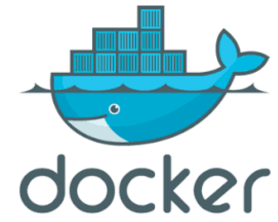
The App should be self-hosted
It should spin up, and start listening on a port, or start talking to one

Apps talk to one another by exposing an address

Avoid hosting in an application server, like IIS



https://docs.microsoft.com/en-us/iis/get-started/introduction-to-iis/introduction-to-iis-architecture



Image

Docker Registry

Docker Host

Container

Docker Daemon

Docker Client

docker run
docker exec
ocker ps

Dockerfile

“Treat backing services as attached resources”

Your ‘backing services’ are all the components and services your app needs to talk to

A database to store data
A message queue to communicate over etc.
But also a file system
And a cache

In cloud, containers, or serverless environments your host is ephemeral.
You cannot rely on its existence across requests

Always assume that you need to ‘provision’ *any* storage that
you want to use

First Demo: A vanilla ASP.NET Core App. Self-Hosted, it listens on a port

Demo

Processes and Concurrency

“Execute the app as one or more stateless processes”

The first key idea here is that your app should be stateless

No sticky sessions, store user’s state in your backing store
Reload your state if required to service subsequent requests

The memory space or filesystem of the process can be used as a brief, single-transaction cache.

Different Workload – Different Process

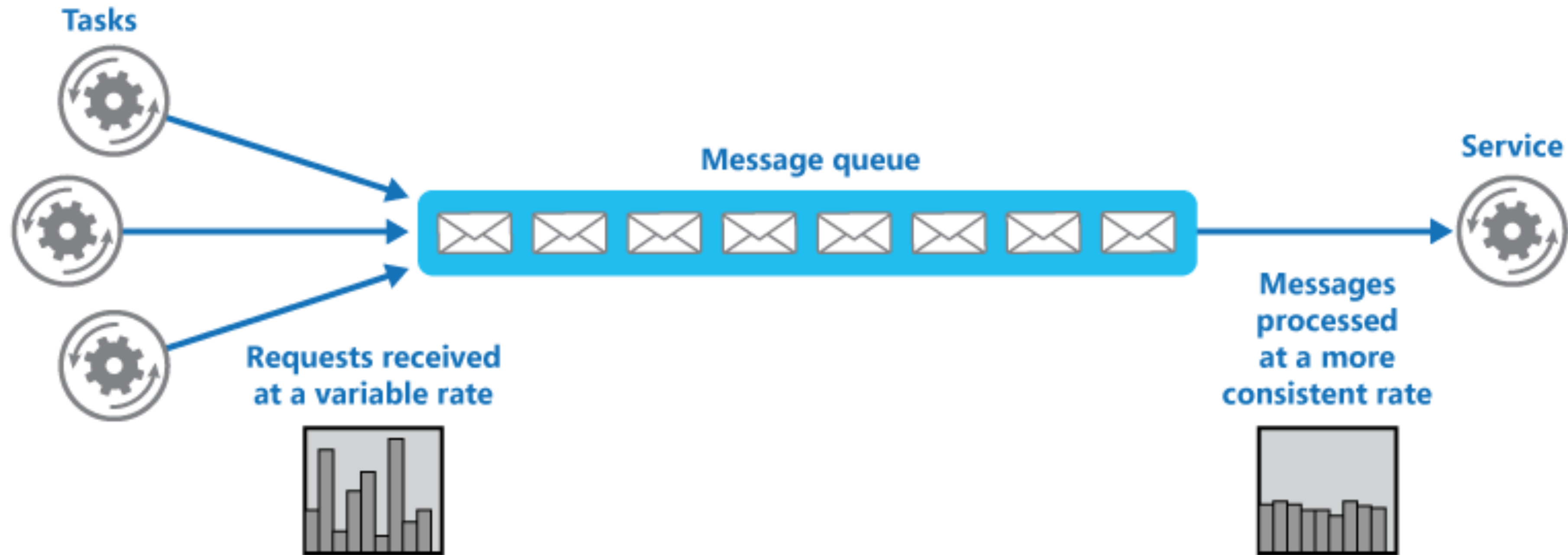
Use separate processes for different workloads

HTTP workloads via a web process; Long-running background tasks by a worker process

An application may experience peaks of demand that cause it to become overloaded and unable to respond.

It could be that part of the application itself becomes overwhelmed or one of its dependencies becomes overwhelmed.

A process that must respond in a timely fashion i.e. for a webserver < 300 ms



https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling

A producer puts a message onto a queue at the service endpoint. A consumer reads messages from the queue.

The queue stores messages for eventual processing. If the queue is durable we gain guaranteed delivery, and at-least once guarantees.

If the rate of arrival at the endpoint is unpredictable, the queue acts as a buffer that makes it possible to predict the rate of consumption.

This makes it simpler to do capacity planning because peaks of requests are smoothed out by the queue.

The consumer must be able to control the rate of processing, otherwise a spike is simply passed down the wire.

Second Demo: A Brighter Worker App, using Console, it listens to AMQP

Demo

“Scale out via the process model”

Prefer to scale out, using new instances of the process

As a process is ‘share-nothing’
We can introduce greater capacity by introducing new processes

Don’t demonize use the operating system’s process manager

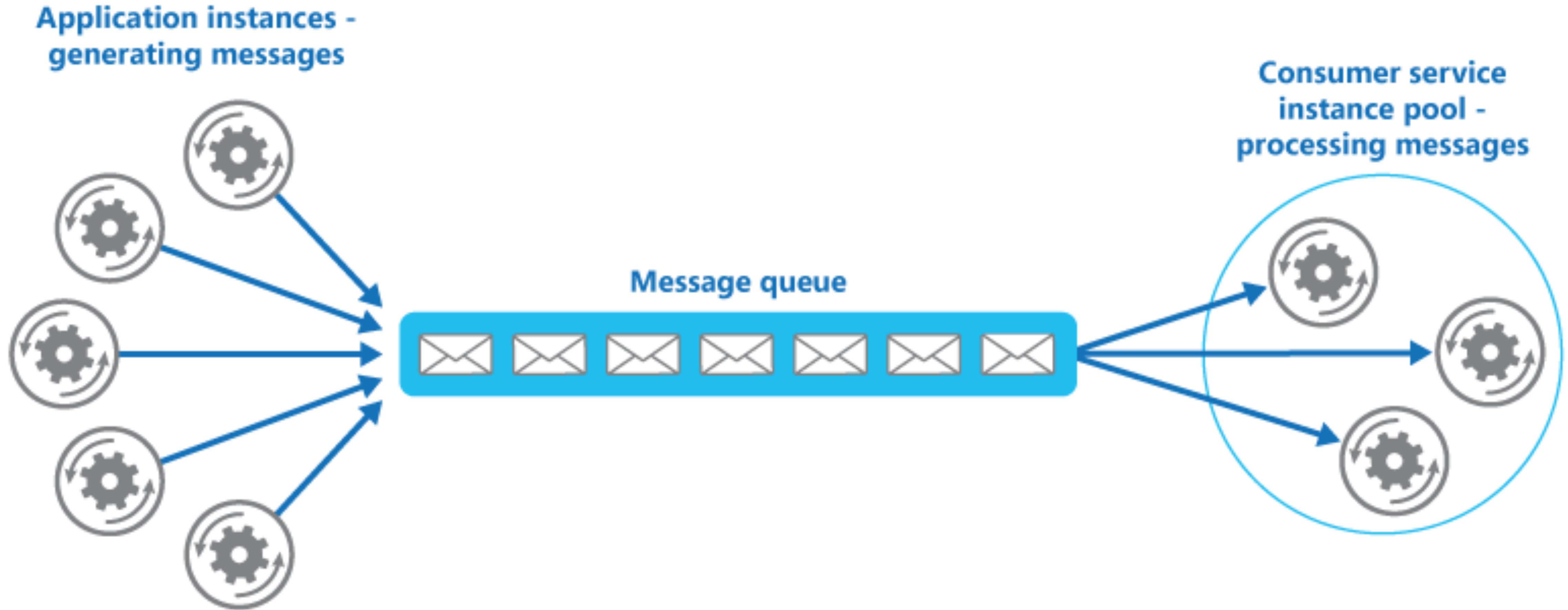
“Maximize robustness with fast startup and graceful shutdown”

A process should be able to start and stop at a moment’s notice

Start in a few seconds

Stop Gracefully

This is needed for elastic scaling, we have to be able to bring new resources online quickly to meet demand or shut them down when done.



https://docs.microsoft.com/en-us/azure/architecture/patterns/competing-consumers

Third Demo: The Docker-Compose File for the App can be used to run multiple instances and load balance between them

Demo

Build & Release

The factors that impact how we deploy applications to the cloud

“One codebase tracked in revision control, many deploys”

The code repository or repo is a copy of the revision tracking control database

1 App == 1 repo
Multiple repos are a distributed system

Shared code must be managed via dependency manager, not shared source in repo

A deploy is a running instance of the app, using the same repo

Demo Prep 1: Pulling the Code

Dependencies

“Explicitly declare and isolate dependencies”

Requires the ecosystem to support a package manager
Requires the ability to isolate an app from other apps i.e. no shared dependencies

Examples include a gemfile or a requirements.txt file produced by Pip
Isolation includes something like Python’s virtualenv

We want to install the runtime and package manager only
clone, restore, build, run

git clone, dotnet restore, dotnet build, dotnet run is a good example of this

Demo

Config

“Store config in the environment”

In this case configuration means anything that varies between deploys

A 12-factor app requires strict separation between config and code.
Config is not checked into the app’s repository.
It often contains secrets such as passwords, or topology dependent information

Because configuration varies by environment – store it in environment variables

Routing tables etc. are not considered configuration for this.
They can often be set in code
But even if you use config, it is not config between enviroments

“Strictly separate build and run stages”

The build stage converts the repo into an executable unit: a *build*
It uses a commit to the repo and fetches dependencies to create an asset

A release is a combination of a *build* and the *config* for a given environment

The run stage executes the app in a given environment

All these stages should be separate.
For example, no changes to deployed code, you have to alter the repo and build again

Demo

Summary

Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project

Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments

Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration

Minimize divergence between development and production, enabling **continuous deployment** for maximum agility

And can **scale up** without significant changes to tooling, architecture, or development practices.

1. One codebase tracked in revision control, many deploys
2. Explicitly declare and isolate dependencies
3. Store config in the environment
4. Treat backing services as attached resources
5. Strictly separate build and run stages
6. Execute the app as one or more stateless processes
7. Export services via port binding
8. Scale out via the process model
9. Maximize robustness with fast startup and graceful shutdown
10. **Keep development, staging, and production as similar as possible**
11. **Treat logs as event streams**
12. **Run admin/management tasks as one-off processes**

Q&A

Please share your feedback

Ian Cooper @ICooper ian@huddle.com