TECH CONFERENCE
DotNet
2020

#DotNet2020

Discovering your superpowers!

Understanding Cloud Application Models

ORGANIZATION

plain concepts

PLATINUM SPONSORS

Microsoft

COLLABORATORS

CEU | Universidad San Pablo

Thank you!

Ryan Nowak

Azure Incubations (Azure CTO's Office)

Formerly: ASP.NET Core architect for MVC/Razor/Blazor

Current Focus: Open Application Model - https://oam.dev/

I like:

- Designing useful tools for developers
- Open Source
- Lots of different languages and technologies

@aVerySpicyBoi

rynowak (github)

Agenda

- What is an Application Model?
- How can we think systematically about cloud runtimes?
- Superpowers!
  - How can we move to production faster?
  - How can we write application code that’s more flexible?

How often do you learn a new technology?

From: Stack Overflow 2020 developer survey

My Journey in 2020

- In the last few years:
  - I've had many conversations with .NET developers about microservices
  - We organize surveys, interviews, and trials for .NET microservices tech
  - Everything *except your code* is a pain point for developers!
- I moved to Azure:
  - Now I'm working on solutions to these problems

What is an Application Model?

A useful definition

An Application Model describes the interface between software components and a runtime environment..

Me

Some Examples

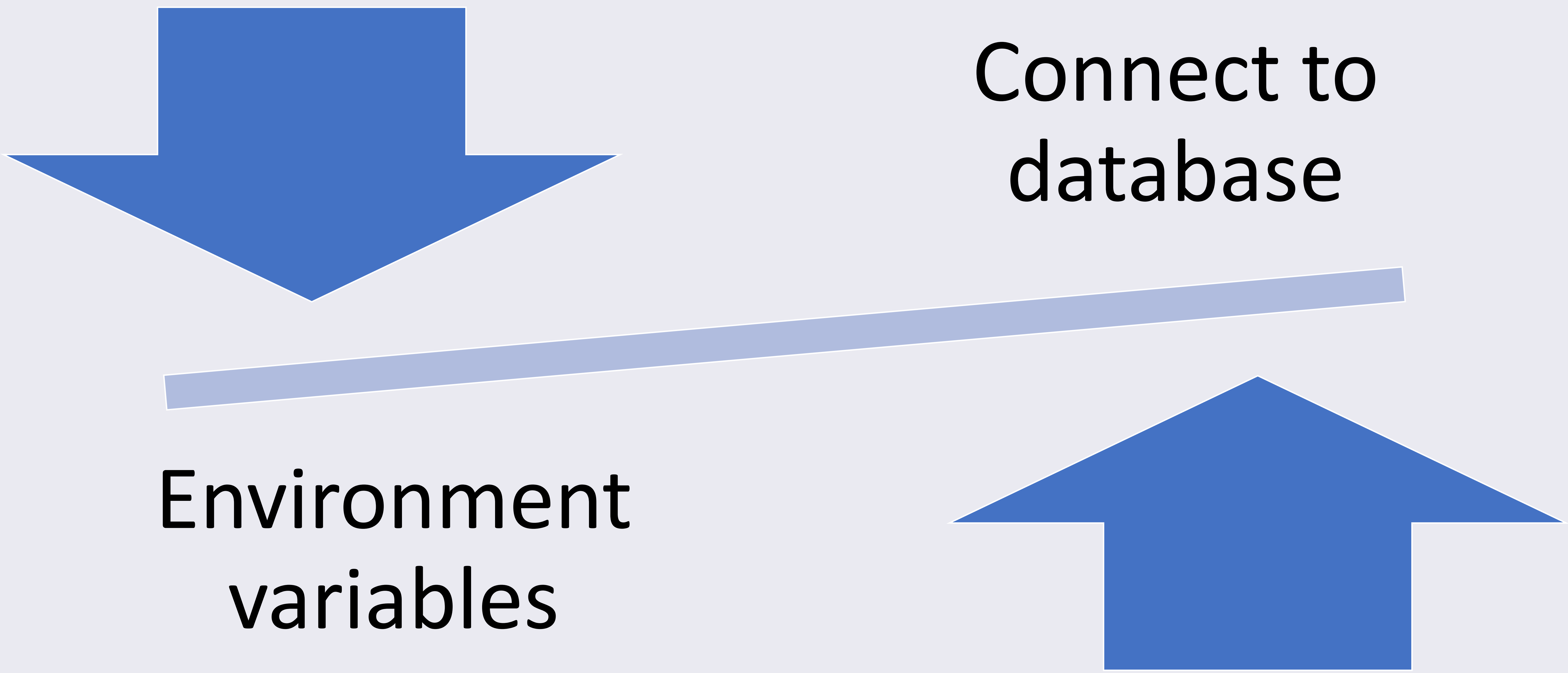
Software Components	Process	"Hosted" Application	Container		
Interface	Environment Variables	Networking	Files	Deployment	Platform behaviors (many more)
Runtime Environment	OS	FaaS (Azure Functions)	Website PaaS (Azure Web Apps)	Docker	Kubernetes

When I talk to developers…

- Learning cloud technologies like Kubernetes or Docker is frustrating
- Development and production are different
- My theories:
  - We deploy as the *last* step … it *works on my machine* and then …
  - We become *users* ... we learn someone else’s software but ...
  - **The concepts we understand don’t match the features offered**

Impedance mismatch

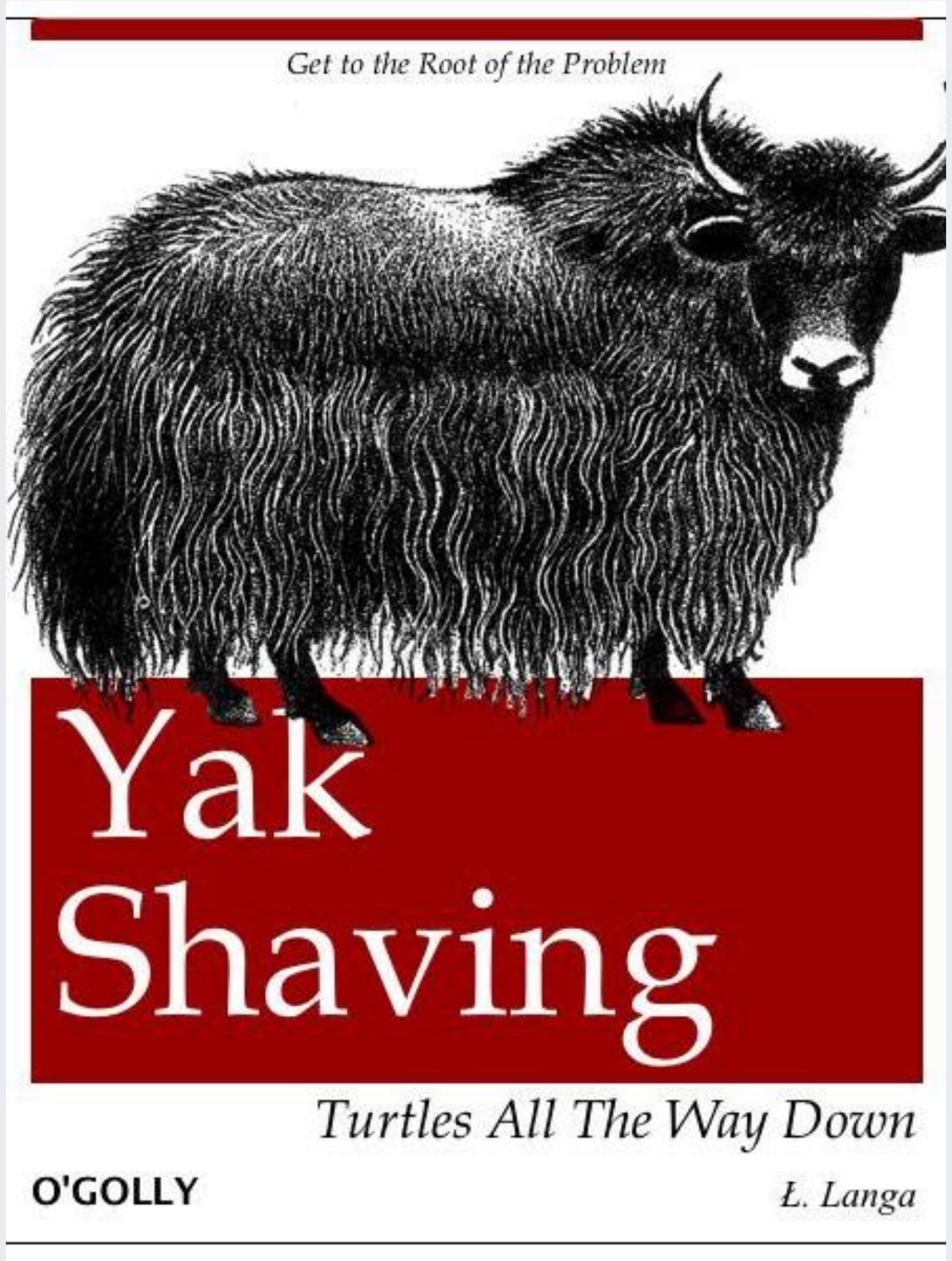
Alternate title: Different levels of abstraction



Yak Shaving

To connect to a database we must:

- Choose an environment variable name
- Figure out the right connection string
- Figure out how to set an environment variable in the deployment platform (without checking in a secret)
- Test it in production (and repeat if you got it wrong)
- Now figure out the next problem and repeat

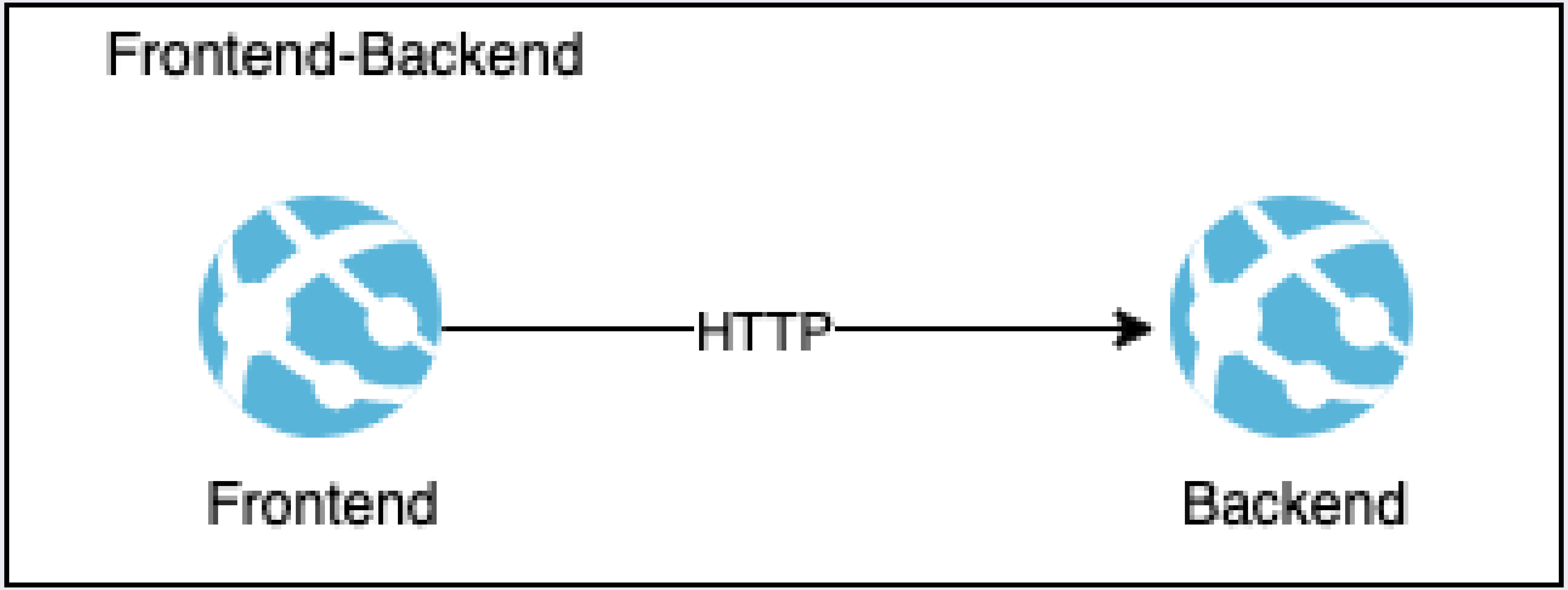


Applying systematic thinking

Planning: For each Service

- What does this service need?
    - What needs to be deployed?
    - Communication with other services? Data stores? Credentials?
    - What settings does it expose?
    - What kinds of diagnostics systems do I need? (Logging at a minimum)
- Understand the capabilities of the platform
    - What options does it provide?
    - What are the tradeoffs of those options?
- Map the needs to the capabilities
    - Write the manifests and deploy!

Planning: Our First App

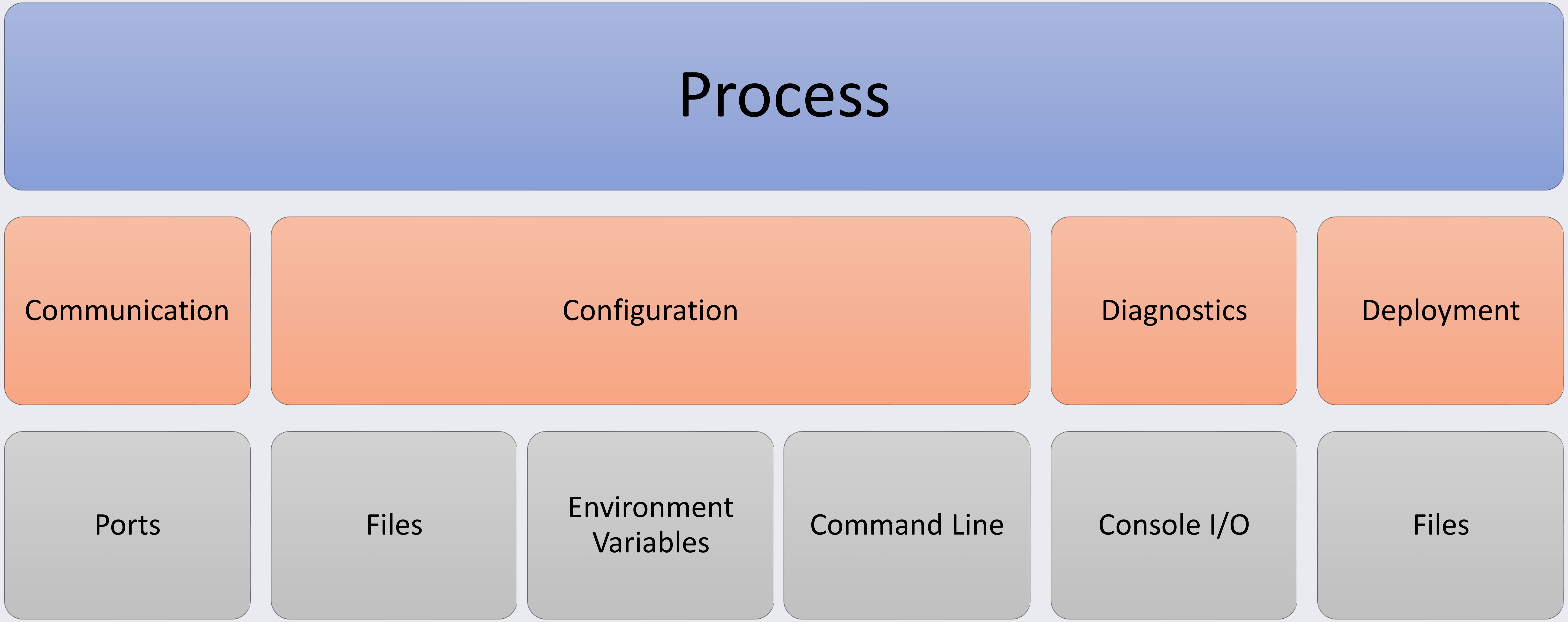


Our First App: What does this service need?

- What needs to be deployed?
  - Both are ASP.NET Core 3.1 applications – so we need that!
- Communication?
  - Backend: Needs to listen for HTTP so frontend can talk to it
  - Frontend: Needs to listen for HTTP so users can browse to it
  - Frontend: Needs to know the address of Backend so it can talk to it
- What settings does it expose?
  - Backend: Listening address,
  - Frontend: Listening address, Address of the backend
- What diagnostics?
  - Logging

Introducing the Process

What did we find?



Being Productive with Processes

Manual management of:

- Ports/URLs
- File Locations

One process is really easy

- 3-4 related processes gets out of control
- You can script it, but it is hard to maintain

We developed Tye to make this easy: https://aka.ms/tye

Something Special: Hosted Environments

You don’t get to start the process - Ex: IIS, Azure Functions

Host runs code in the same process as yours

- Usually this comes with an SDK

Networking is controlled by the host

- You don’t choose a port to listen on

The host provides a deployment format

- This is usually a zip file plus a manifest

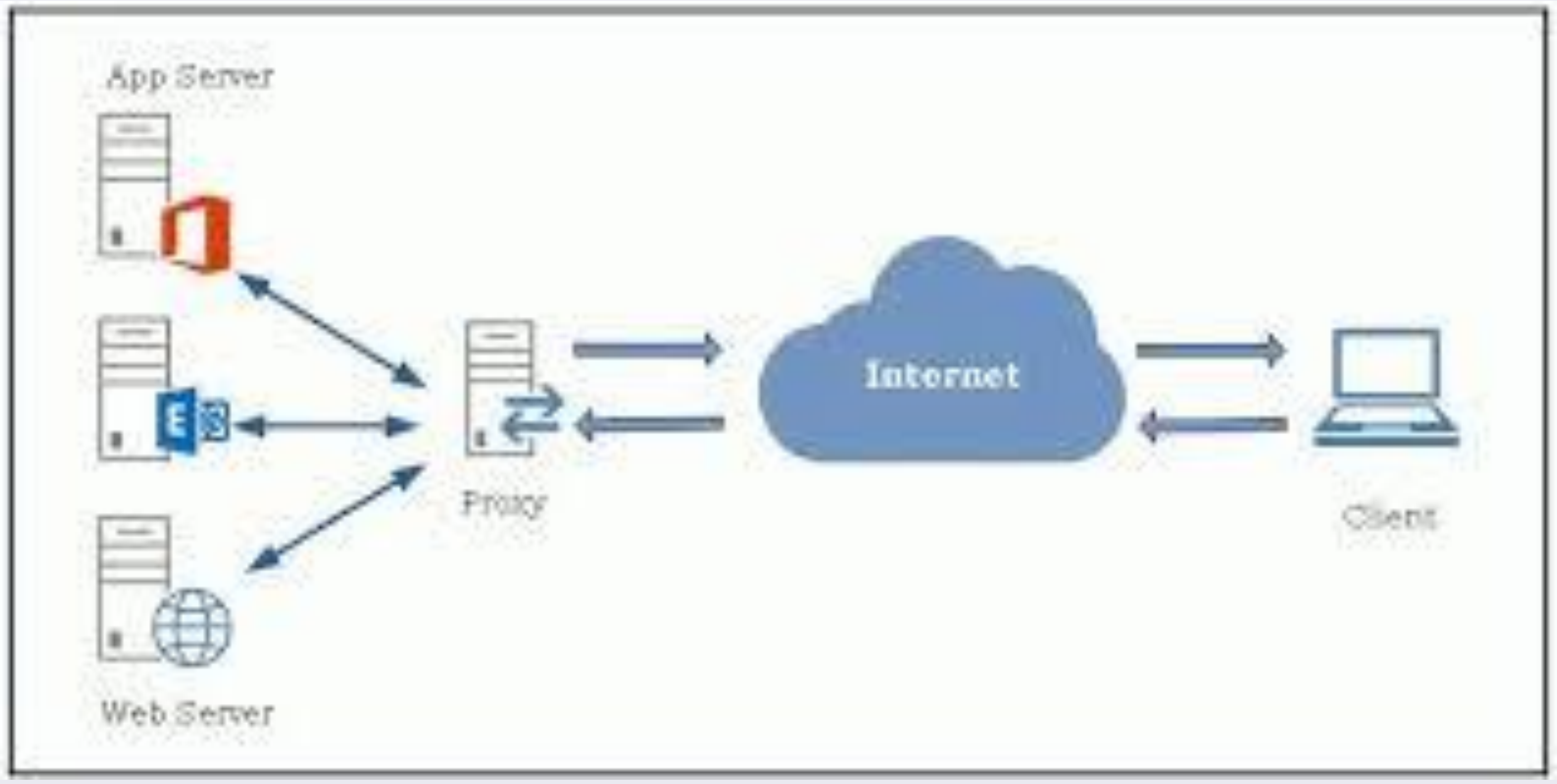
Hosted Environments

Host

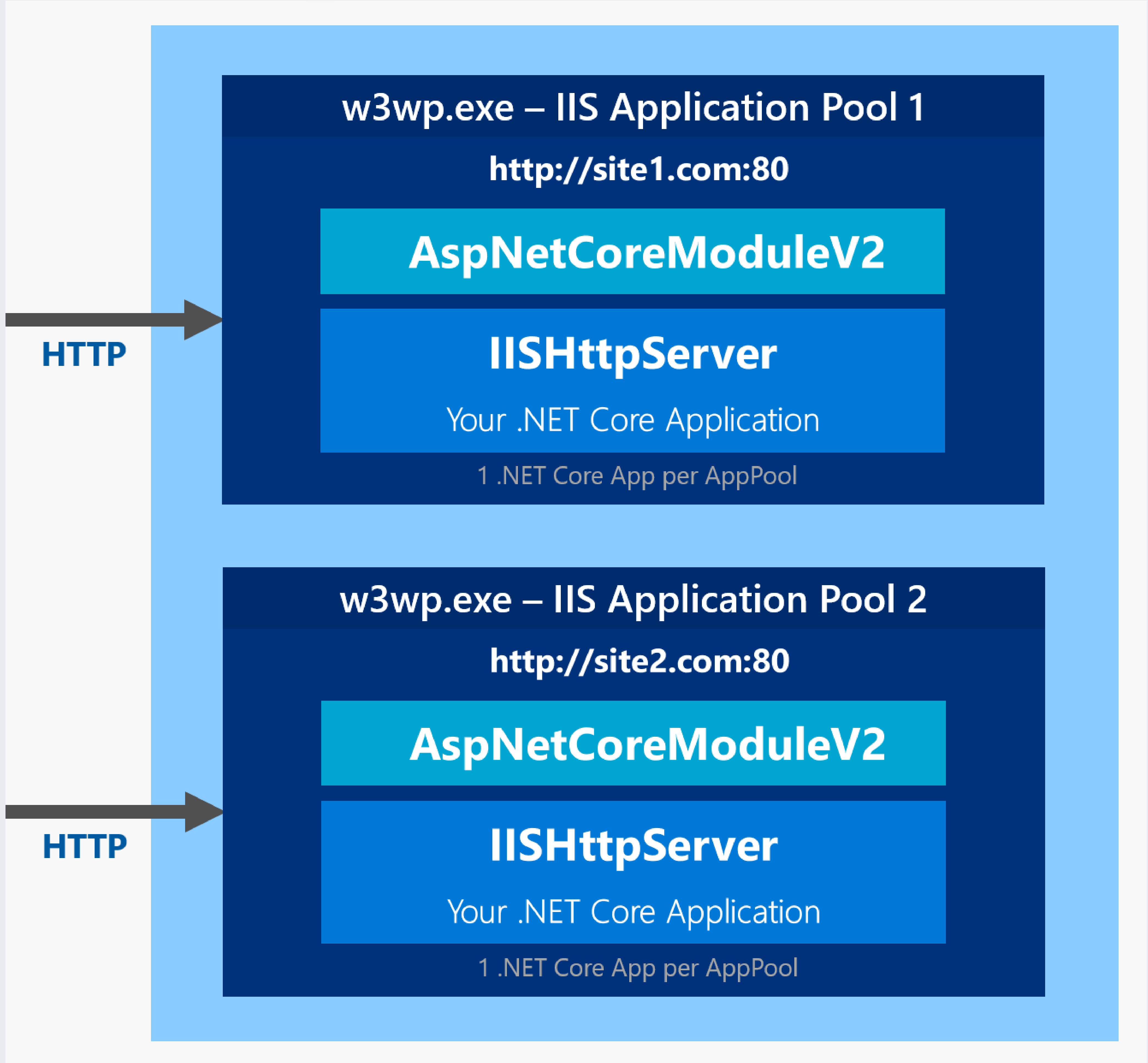
Communication
Configuration
Diagnostics
Deployment

Ports (managed by host)
Other Bindings
Host Configuration
Files (maybe)
Logging API (managed by host)
Files

Two models



Reverse Proxy



In-Process Hosting

Two Models

Reverse Proxy

Communication happens over sockets

Your application could be *anything*

You choose an address and configure the server to point at it

No SDK for interacting with proxy in code (you figure out diagnostics, lifecycle)

In-Process Hosting

Communication happens through API

Your application must be a supported platform of the server

Server manages address, configuration

Comes with SDK for diagnostics, lifecycle, rich API

Ex: IIS, Azure WebSites, Tomcat

Two Models

Old ASP.NET (.NET Framework) uses the In-Process model and is coupled to IIS & Windows

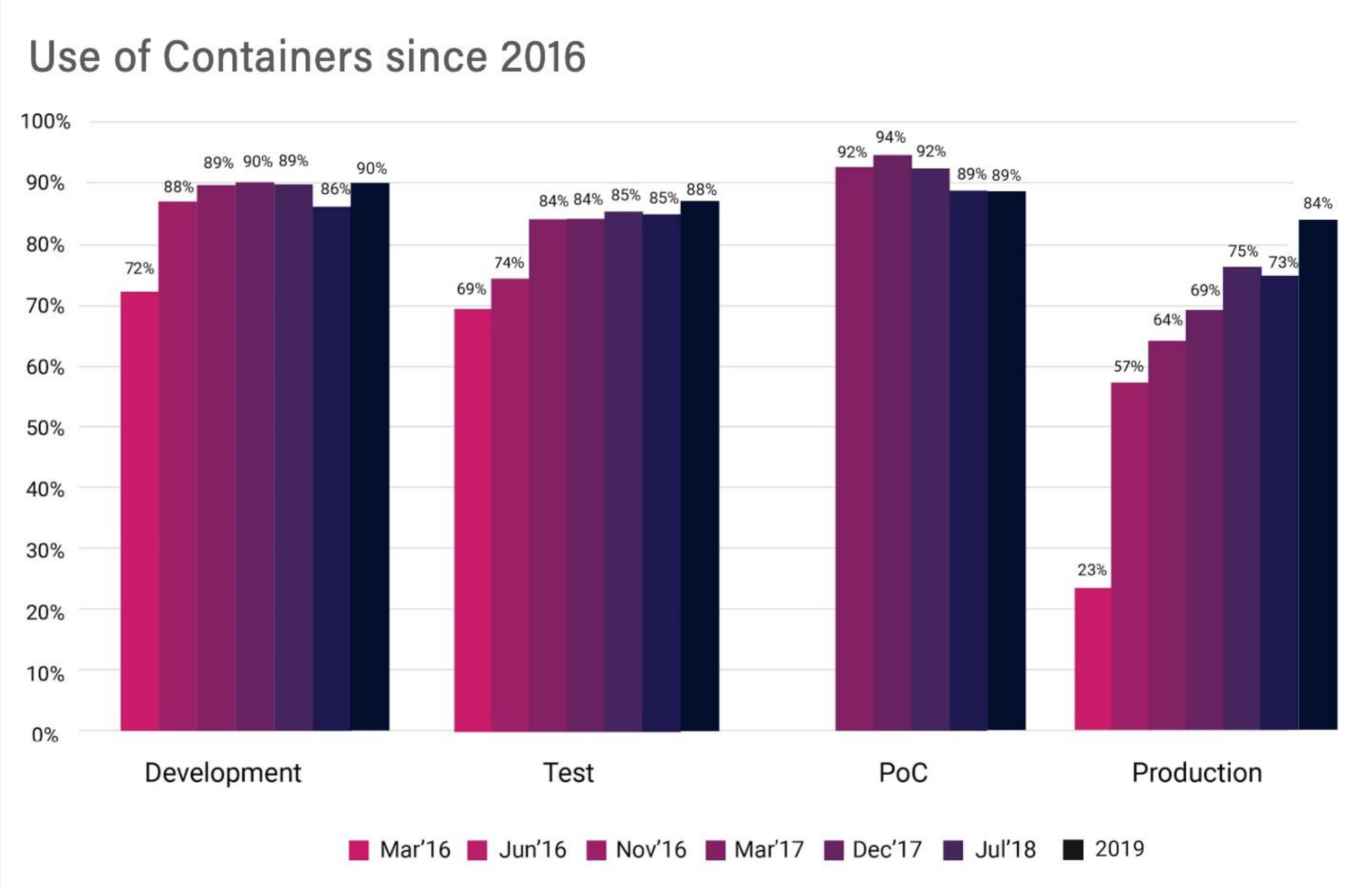
- System.Web.dll is a tightly coupled to IIS's architecture and API
- Some projects of that generation supported non-IIS (Katana)

ASP.NET Core used the Reverse Proxy architecture on release

- Kestrel server was first based on LibUV (used in node) as a cross-platform standalone server
- We made heavy investments in System.Net.Socket and far surpassed LibUV over the years
- IIS has a module for Reverse Proxy integration (used in Azure WebSites)
- This was slow and diagnostics were bad compared to ASP.NET Core without IIS
- We implemented In-Process support in 2.2 without coupling ASP.NET Core to Windows

And then for a long time nothing happened…

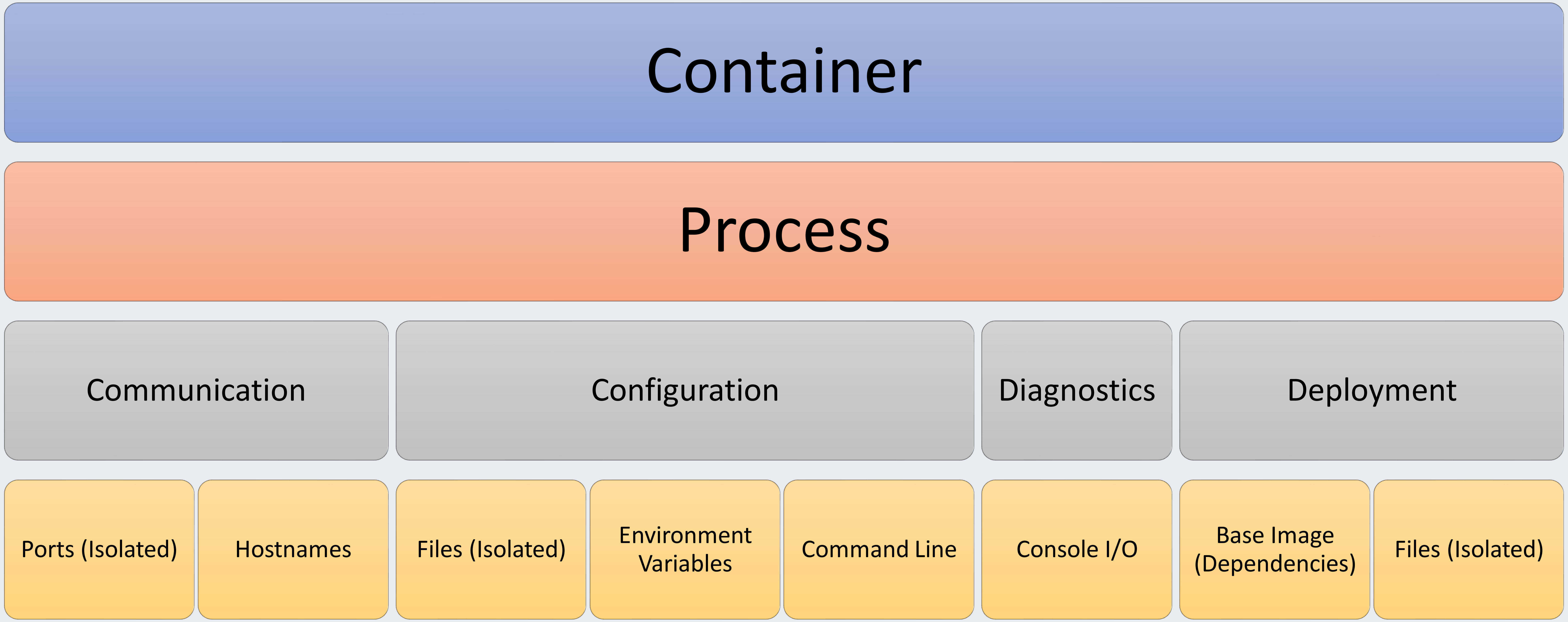
Introducing Containers



CNCF survey

Introducing Containers

What did we find



Being Productive with Containers

Ports and files are now isolated

- Now we have consistency instead of random numbers to remember

Can bake configuration into the image

- Files and startup command are part of the image
- Environment variables still useful for overrides

Deployment is more powerful and foolproof

- Versioning/naming of images
- Ability to use registries for storage
- Can ship arbitrary dependencies and OS configuration in image

This sounds great right?

Except Dockerfiles are really painful ☹

- Easy to copy-paste and therefore a lot of copies to maintain
- Usually 1-2 people on the team really know how it works
- Optimizing your Docker build for size increases the complexity
- 2-phase build makes it hard to use P2P references or shared build assets
- I don't know anyone who thinks this is easy and fun…

My controversial advice

Think hard about what you want to optimize for

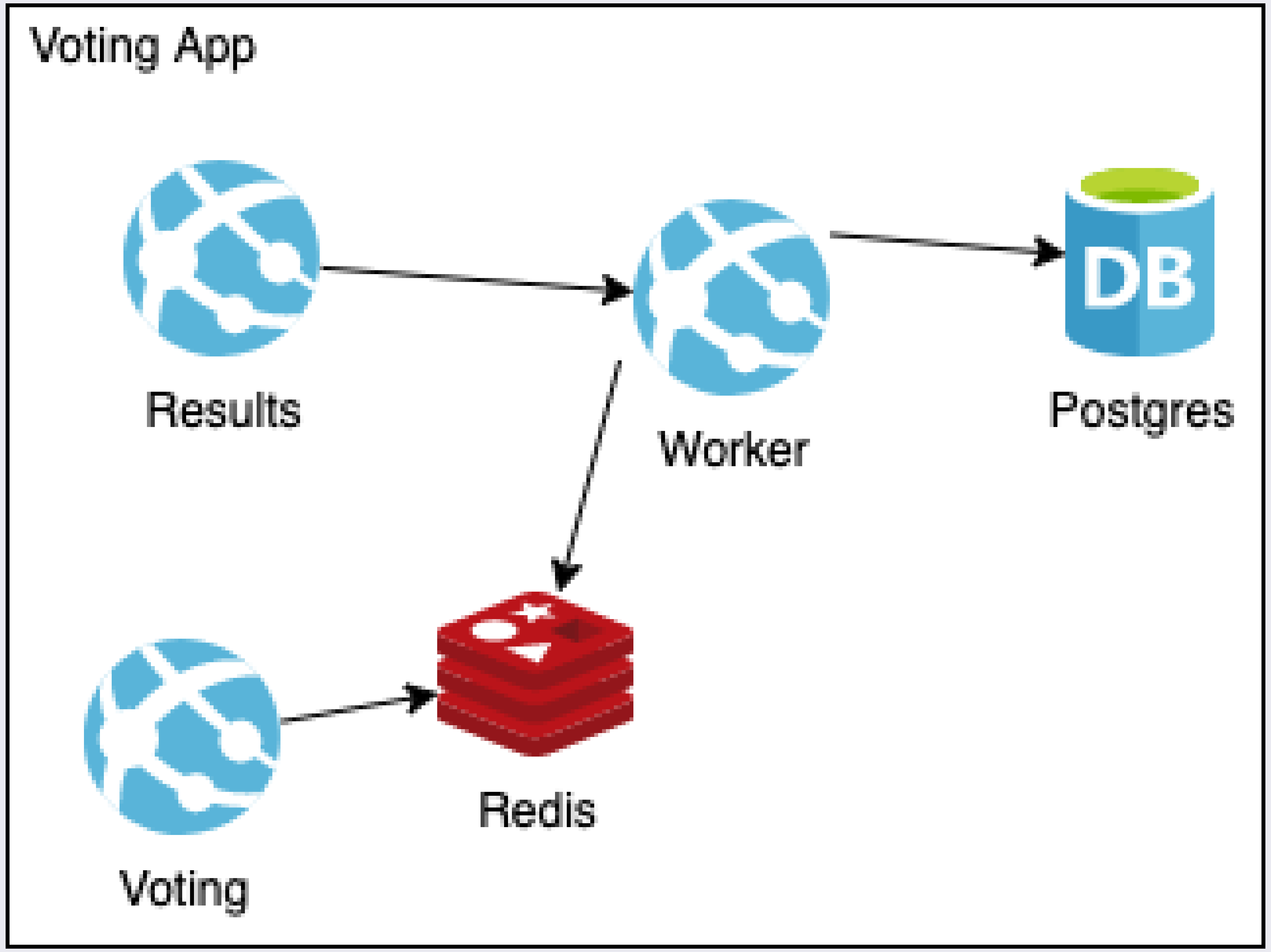
- Don’t copy techniques from native-code platforms for “best practices” reasons
- Optimize for maintainability until you need to optimize for something else

Recipes that work:

- (service-per-repo) Copy whole repo into Container and build (2-phase) inside – very flexible
- (few dependencies) Publish files with a script and build with CLI (1-phase) – no Dockerfiles!
- (smallest total size) Use the linker with a standalone publish – final layer will be big but total size small

More than one at a time

Planning: A more complex app



What does this service need?

- What needs to be deployed?
  - All are ASP.NET Core 3.1 applications – so we need that!
- Communication?
  - Voting: needs to connect to redis
  - Worker: needs to connect to redis and postgres
  - Results: needs to connect to worker
  - All need to listen on HTTP: voting and results for users, worker for other services
- What settings does it expose?
  - Voting: list of categories is configurable
  - All: URLs and Connection Strings
- What diagnostics?
  - Logging

Docker's capabilities

- Deployment:
  - Containers!
  - Can configure docker-compose to build images for us
- Communication:
  - We need to map a port per externally available service
  - Docker provides port isolation for internal communication
  - Docker will assign a hostname for internal communication
- Settings:
  - Docker has all standard things (files, command line, environment variables)
  - Docker also has a secret store which get mounted as files in the container (not used here)
- Diagnostics
  - Logging via Console I/O

We already understand Docker…

- Deployment:
  - Use the ASP.NET Core base image
  - Note: we're running redis and postgres as images for local/dev puposes
- Communication:
  - Use port 80 (default for HTTP) since we have port isolation
  - Use environment variables to configure URLs relying on docker's hostnames
  - Use environment variables to pass connection strings (hardcoded in docker-compose.yaml for dev)
- Settings:
  - Using files inside the images and environment variables
- Diagnostics:
  - Using console logging (default)

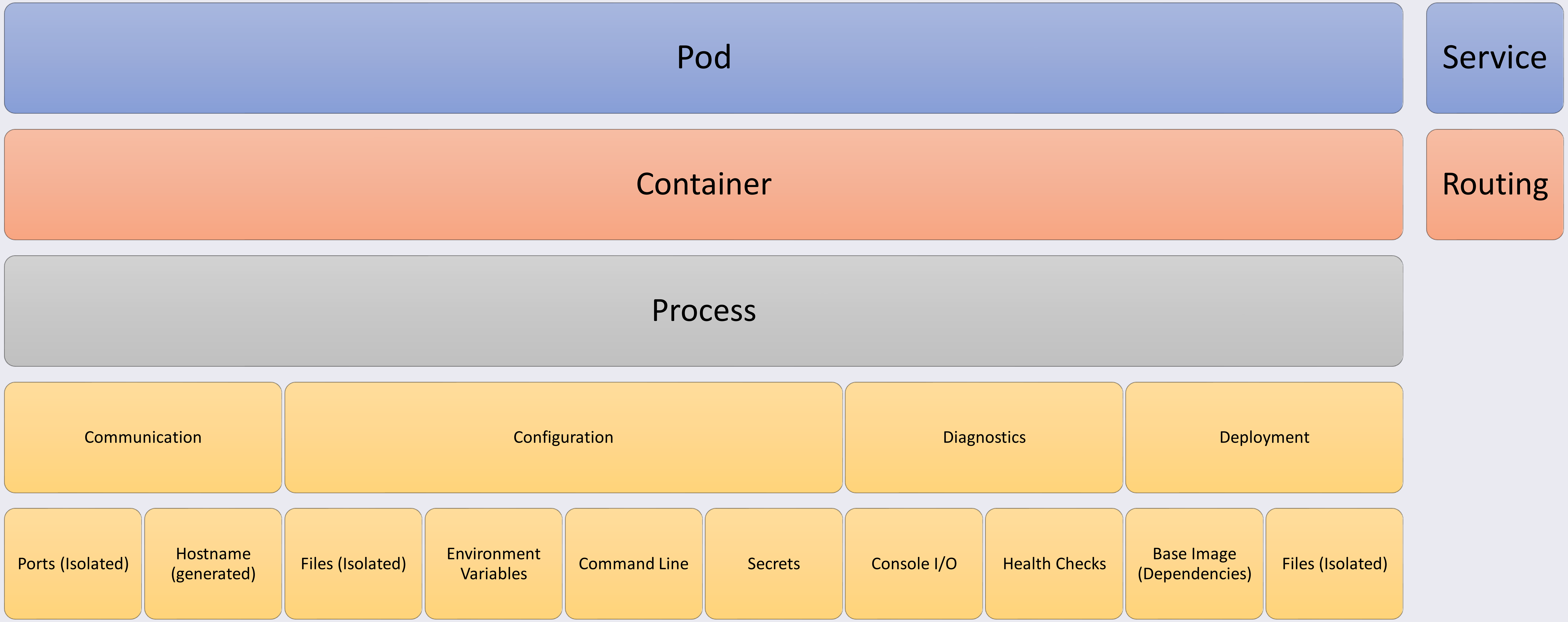
Docker-Compose

Kubernetes

A very fast Kubernetes primer

- Kubernetes has lots of features and lots of types
  - Deployment is the one you want to for your services
  - The type system in Kubernetes contributes a lot of complexity to the format
- Kubernetes objects have:
  - Metadata (name, namespace, labels, more…)
  - Status (data tracked by the runtime)
  - Spec (details you configure)
  - The use of the same object for all of this contributes even mode complexity
- Kubernetes has powerful features for routing and networking (Service)
  - Routing requires a separate object from the deployment
  - Thus, we need a way to indicate relationship between Deployments and Services
- This is enough to scratch the surface – many of the concepts in Kubernetes are for Ops not Developers

What did we find



Wrap Up

Application Models

- "An Application Model describes the interface between software components and a runtime environment"
- Your job is to use the model provided by your runtime to describe your application
- Application models vary tremendously in richness and what concepts they use
- I work on Open Application Model (OAM) – https://oam.dev to make life better

Practical Applications: Design

- Apply systematic thinking
  - What does each service need?
    - Deployment
    - Communication
    - Configuration
    - Diagnostics
  - Understand your platform:
    - What options does the platform provide?
    - How does your platform work? (hosted vs proxy) (container vs process)
  - Map your needs onto what the platform supports
    - This is ultimately what you need to write down to deploy successfully!

Practical Applications: ASP.NET Core

- Leverage what ASP.NET Core provides:
  - Configuration
  - Logging
  - Server Configuration
- Don’t hardcode! Configure:
  - Listening Ports
  - URLs/Connection Strings
- Check out Tye for multi-service development: https://aka.ms/tye

What I didn’t cover…. More diagnostics

- Robust diagnostics is the difference between a demo and a real application ☺
- These aren’t usually tied to the runtime environment
- There are more types of logging systems out there:
  - Structured Logging with JSON
- Metrics
- Distributed Tracing

DotNet2020

#DotNet2020

Thanks also to the sponsors.
Without whom this would not have been posible.

Thanks and ...
See you soon!