

TWELVE FACTOR APPS

methodology for building software-as-a-service apps

THE TWELVE FACTORS

1. Codebase
2. Dependencies
3. Config
4. Backing services
5. Build, release, run
6. Processes

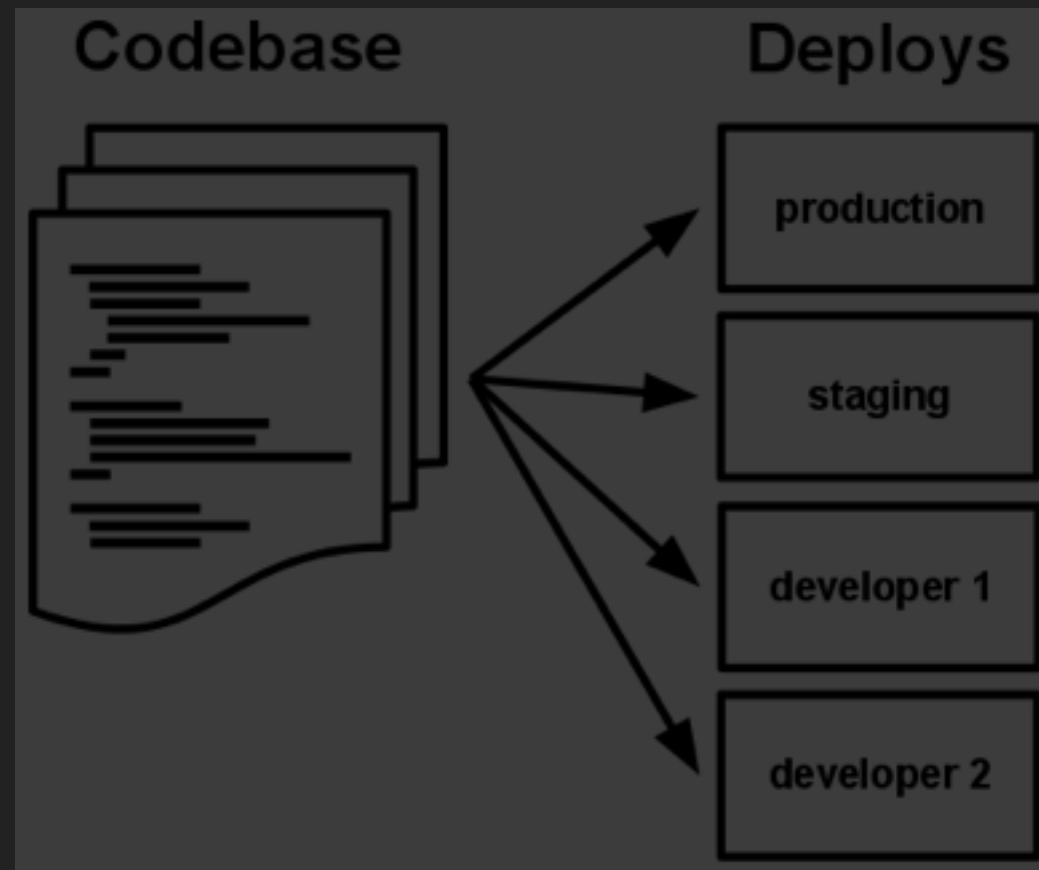
THE TWELVE FACTORS

- 7. Port binding
- 8. Concurrency
- 9. Disposability
- 10. Dev/prod parity
- 11. Logs
- 12. Admin processes

1. CODEBASE

- code is tracked in a version control system (e.g. Git)
- codebase is single repository
- one codebase per app
- multiple apps share code via libraries

A deploy is a running instance of the app



2. DEPENDENCIES

- explicitly declare and isolate dependencies
- never rely on implicit existence of system-wide packages

3. CONFIG

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

CONFIG VS. CODE

- strict separation of config from code
- config varies substantially across deploys, code does not

CONFIGURATION FILES

Disadvantages:

- easy to mistakenly check in to the repo
- tendency to be scattered about in
 - different places
 - different formats

ENVIRONMENT VARIABLES

store config in environment variables

- easy to change between deploys without changing any code
- not accidentally checked into repo
- language- and OS-agnostic standard
- don't name after specific deploys

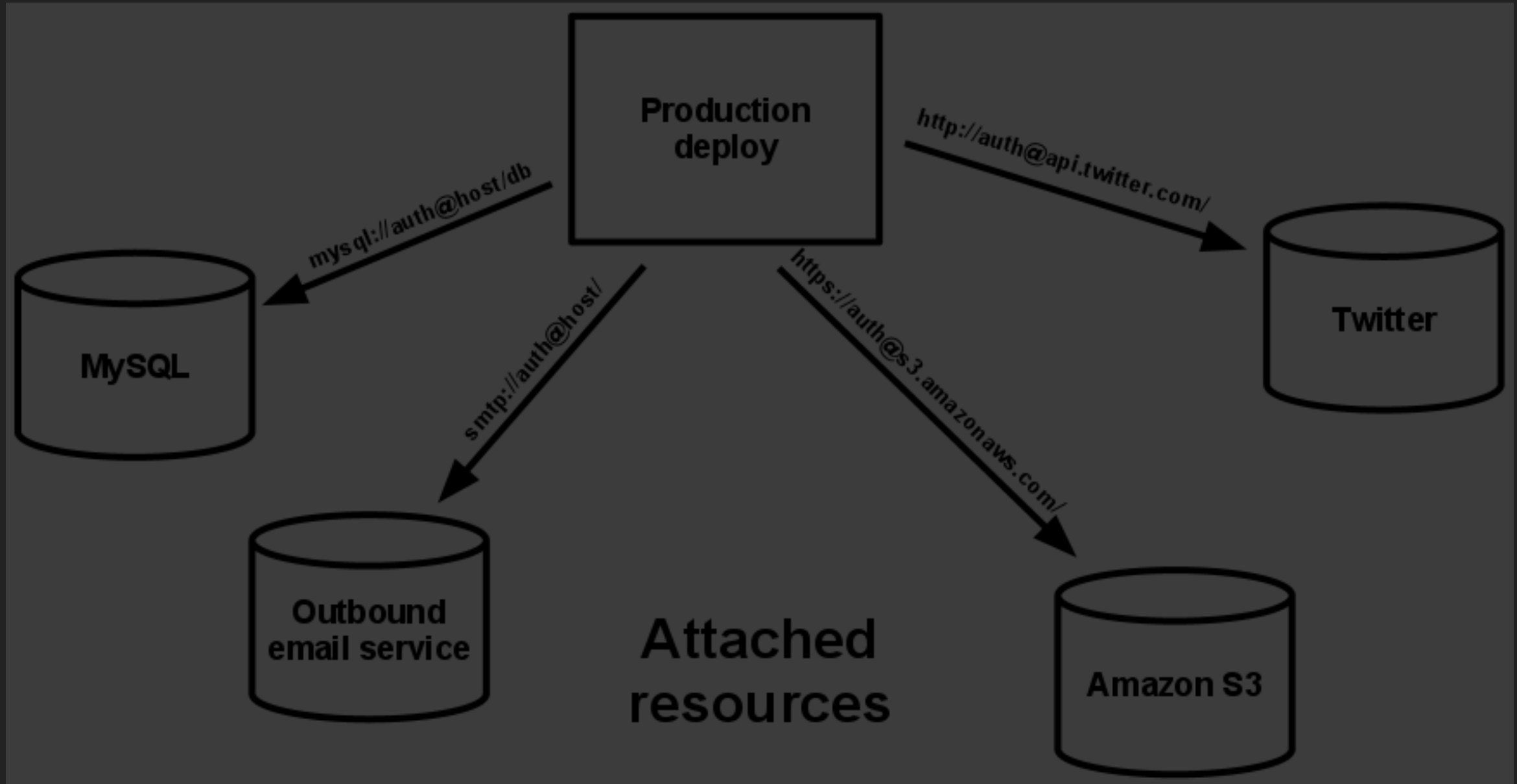
ENVIRONMENT VARIABLES CONSIDERED HARMFUL FOR YOUR SECRETS

When you store your secret keys in the environment, you are still prone to accidentally expose them -- exactly what we want to avoid.

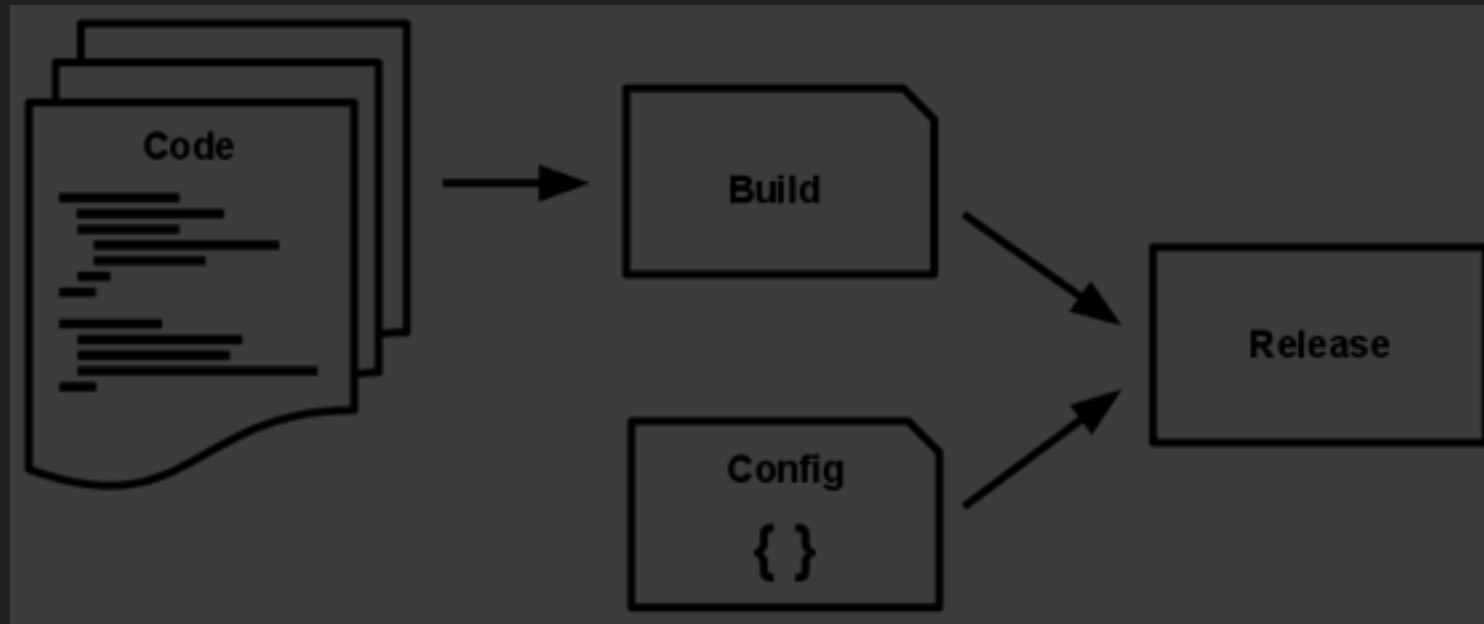
VAULT – A SECRET STORE

- stores credentials encrypted
- provision credentials dynamically
- completely free and open source

4. BACKING SERVICES



5. BUILD, RELEASE, RUN



strict separation between stages:

1. **build** = code repo converted into executable bundle
2. **release** = build + config
3. **run** release in the execution environment

RELEASES

- every release has a unique ID
- any change must create a new release

6. PROCESSES

- app is executed in the execution environment as one or more processes
- processes are stateless and share-nothing
- persist data using stateful backing service (DB)
- memory space or filesystem as brief, single-transaction cache

7. PORT BINDING

- app is completely self-contained
- export services via port binding
- app can become the backing service for another app

8. CONCURRENCY

- scale out via the process model
- processes are a first class citizen
- app must be able to span multiple processes running on multiple physical machines

9. DISPOSABILITY

- fast startup and shutdown
- shut down gracefully when receiving a **SIGTERM**
 - refuse new requests
 - allow current requests to finish

CRASH ONLY SOFTWARE

Stop = Crash Safely

Start = Recover Fast

10. DEV/PROD PARITY

	Traditional	12-factor
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
- write unbuffered to `stdout`
- execution env. completely manages streams

LOGGING TOOLS

- log routers: [Logplex](#), [Fluentd](#)
- log indexing and analysis system: [Splunk](#)
- general-purpose data warehousing system:
[Hadoop/Hive](#)

12. ADMIN PROCESSES

ONE-OFF ADMINISTRATIVE OR MAINTENANCE TASKS

- database migrations
- console to run arbitrary code
- one-time scripts committed into the app's repo

HOW TO RUN

- identical environment
- run against a release
- use the same codebase and config

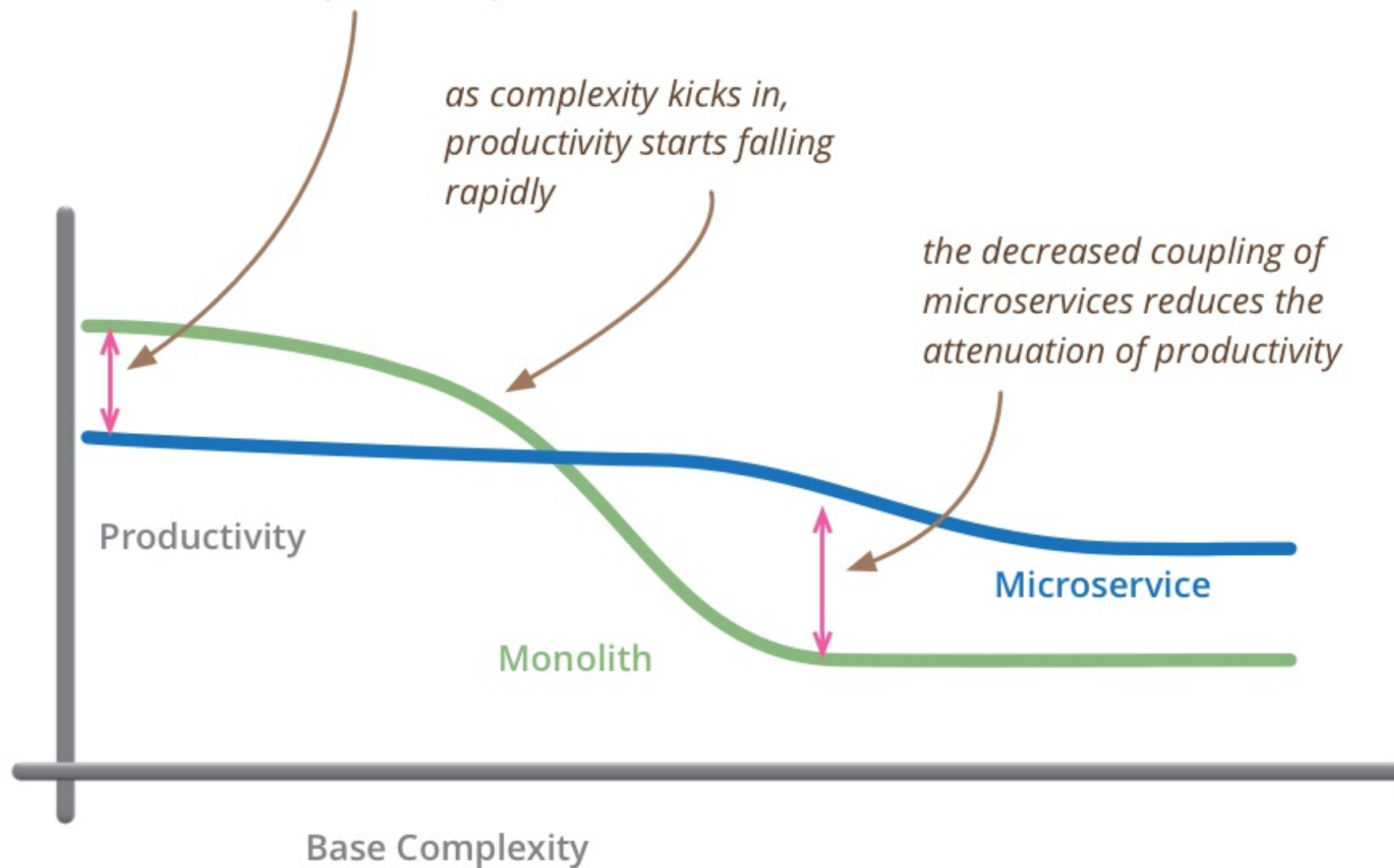
SHOULD I USE MICROSERVICES?

*any decent answer to an interesting
question begins, "it depends..."
(Kent Beck)*

WHEN YOU USE MICROSERVICES YOU HAVE TO WORK ON

- automated deployment
- monitoring
- dealing with failure
- eventual consistency
- other factors that a distributed system introduces

for less-complex systems, the extra baggage required to manage microservices reduces productivity



but remember the skill of the team will outweigh any monolith/microservice choice

*don't even consider microservices
unless you have a system that's too
complex to manage as a monolith*

Martin Fowler

The End.