GitOps

Beyond Patterns and Principles

Joel Bennett

April 8-11, 2024

POWERSHELL DEVOPS GLOBAL SUMMIT

Thanks as always to our sponsors!

Joel "Jaykul" Bennett

Principal DevOps Engineer

```
Solving problems with code
15x Microsoft MVP for PowerShell
```

github.com/Jaykul and PoshCode
discord.gg/PowerShell
HuddledMasses.org
@Jaykul@FOSStodon.org

POWERSHELL DEVOPS GLOBAL SUMMIT

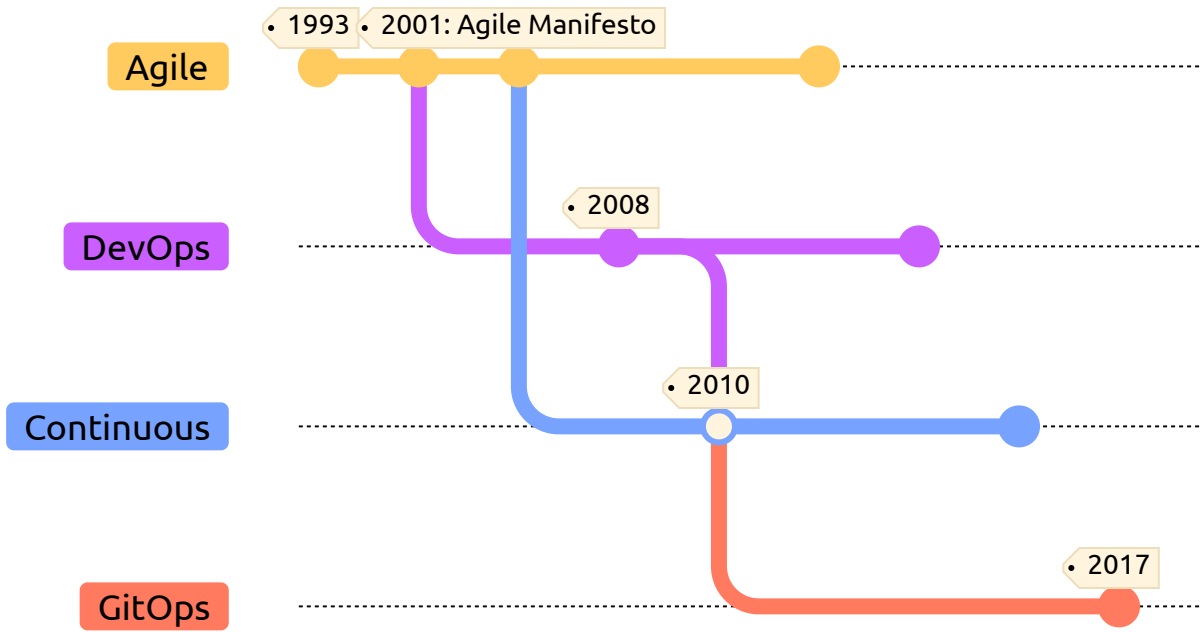
What is GitOps?

Principles for Operating and Managing Software Systems

We are not merely building infrastructure from code

We seek to create *systems* that continuously update and repair themselves

Historical Context



Historical Context

Agile Software Development (2001)

The *agile manifesto* prioritized collaboration and flexibility…

DevOps (2008)

DevOps centers on cultural change, unifying Dev and Ops organizations with common goals and shared KPI.

Continuous Delivery (or Deployment) (2010)

Continuous Delivery became a noun, with a focus on automated pipelines, and small, frequent releases.

GitOps (2017)

GitOps is a *set of principles* for operating and managing software systems.

GitOps Principles

v1.0.0

1 Declarative

A system managed by GitOps must have its desired state expressed declaratively.

2 Versioned and Immutable

Desired state is stored in a way that enforces immutability, versioning and retains a complete version history.

3 Pulled Automatically

Software agents automatically pull the desired state declarations from the source.

4 Continuously Reconciled

Software agents continuously observe actual system state and attempt to apply the desired state.

Declarative

The code for the desired state is *independent* of the steps needed to get there.

Versioned and immutable

It’s code all the way down

1. The software *source* is code
2. The infrastructure *source* is code
3. The configuration *source* is code
4. *All **source** code is version controlled*

Pulled Automatically

GitOps uses **software agents** that **pull** the desired state from source. This scales well, and only requires *read* access to source.

Continuously Reconciled

Reconciliation is the process of ensuring the *actual state* matches the *desired state*. It's triggered *whenever* there is a divergence – whether a new version of the desired state, or drift in the actual state.

Continuous Deployment is great

But please don't conflate

GitOps wants to be more

Conceptually different

> GitOps follows control theory and operates in a closed-loop, where feedback represents how previous attempts to apply a desired state have affected the actual state. Actions are taken based on policies, it order to *reduce deviation* over time.

- Is the Octopus Deploy Agent a GitOps agent?
- Is Windows Desired State Configuration (DSC) a GitOps agent?

Tooling

- Infrastructure As Code
- Continuous Delivery
- GitOps Agents

Infrastructure As Code

The Code Should be *Declarative*

The Tools Should be *Idempotent*

- Terraform by Hashicorp, and OpenTofu by Linux Foundation
- Pulumi supports Yaml – but also C#, Python, Go, TypeScript.
- Idem by VMWare (infrastructure as "data" = yaml)
- Radius does Bicep **and** Terraform
- Bicep / Arm / CloudFormation

Continuous Delivery

Build, Test, Package

Every source-hosting platform has a CI/CD system…

- GitLab
- GitHub Actions
- Azure Pipelines

And there are many, many others.

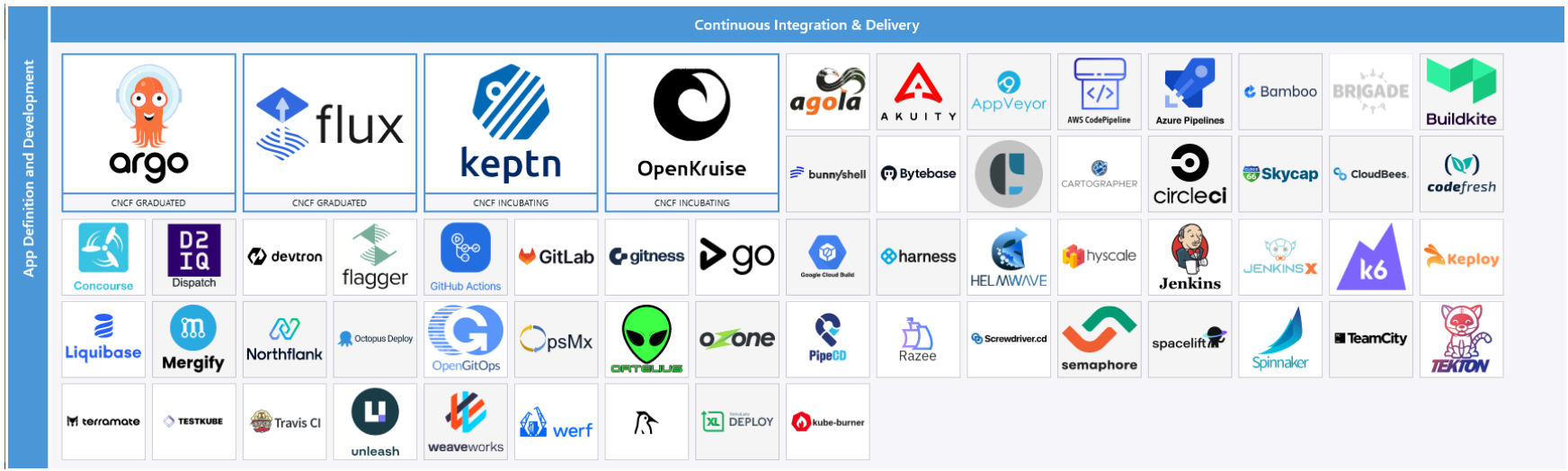
- Bamboo (BitBucket)
- CircleCI
- TeamCity
- Travis CI
- Jenkins

Some even run in Kubernetes:

- GitLab **Agent**
- Tekton
- Jenkins X
- Skaffold

GitOps Agents

- Flux
- ArgoCD
- Fleet



Show and Tell

- AKS-Bicep
- Cluster

Best Practices

Or, how do I GitOps?

Standardization

Twelve-Factor Apps

Change Management

Standardization

In short, you can't control a train or a ship with the autopilot for a Tesla.

Especially with micro-services

Automation requires standardization

Reduce variations and alternatives

Avoid custom one-off implementations

Standardization

Start small

- Source control
- Pull requests
- Continuous **Delivery**
- Automate Deployments
- Health checks

Best of breed tools

- Code-based
- Declarative
- Idempotent

Dependency management

- Vulnerability scanning
- Prevent stale
- Included in Health Checks
- Included in Auto Scaling

Best Practices

The Twelve-Factor App (12factor.net)

I. Codebase

One codebase tracked in revision control, many deploys

II. Dependencies

Explicitly declare and isolate dependencies

III. Config

Store config in the environment

IV. Backing services

Treat backing services as attached resources

V. Build, release, run

Strictly separate build and run stages

VI. Processes

Execute the app as one or more stateless processes

VII. Port binding

Export services via port binding

VIII. Concurrency

Scale out via the process model

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity

Keep development, staging, and production as similar as possible

XI. Logs

Treat logs as event streams

XII. Admin processes

Run admin/management tasks as one-off processes

Change Management

Separate Repositories

Infrastructure != Application != Configuration

- Independent life-cycles
- Separate approval processes
- Different team ownership

Environment Configuration

Don’t use mono-repos or branch-per-environment

- Create versioned configuration artifacts
- Import artifacts, or inherit a shared base
- Use overlays for different environments
- Think about versioning and promotion of the base

Background

Agile Manifesto (.org)

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more

Principles behind the Agile Manifesto (1/2)

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Business people and developers must work together daily throughout the project.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Principles behind the Agile Manifesto (2/2)

Working software is the primary measure of progress.

Simplicity–the art of maximizing the amount of work not done–is essential.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

The best architectures, requirements, and designs emerge from self-organizing teams.

Continuous attention to technical excellence and good design enhances agility.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Best Practices

The Twelve-Factor App (12factor.net)

I. Codebase

One codebase tracked in revision control, many deploys

II. Dependencies

Explicitly declare and isolate dependencies

III. Config

Store config in the environment

IV. Backing services

Treat backing services as attached resources

V. Build, release, run

Strictly separate build and run stages

VI. Processes

Execute the app as one or more stateless processes

VII. Port binding

Export services via port binding

VIII. Concurrency

Scale out via the process model

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity

Keep development, staging, and production as similar as possible

XI. Logs

Treat logs as event streams

XII. Admin processes

Run admin/management tasks as one-off processes

1. Codebase

There is always a one-to-one correlation between the codebase and the app

- If there are multiple code bases, it's not an app – it's a distributed system.
- Each component in a distributed system is an app, and each can individually comply with twelve-factor.
- Multiple apps sharing the same code is a violation of twelve-factor.
- Factor shared code into libraries which can be included through the dependency manager.

A deploy is a running instance of the app.

There is only one codebase per app, but there will be many deploys of the app: production, staging, etc.

The codebase is the same across all deploys, although different *versions* may be active in each deploy.

2. Dependencies

Explicitly declare and isolate dependencies

A twelve-factor app never relies on implicit existence of system-wide packages or system tools.

It **declares all dependencies**, completely and exactly, via a dependency declaration manifest, and uses **dependency isolation** to ensure no implicit dependencies "leak in" from the surrounding system.

One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app.

3. Config

Store config in the environment

An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, cache, 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

One litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

The twelve-factor app stores config in environment variables. These are easy to change, and hard to accidentally commit. They are a language- and OS-agnostic standard.

These variables are not grouped into sets or environments, but instead are granular and managed ndently for each deployment.

4. Backing Services

Treat backing services as attached resources

A backing service is any service the app consumes over the network as part of its normal operation.

Examples include:

- Datastores (such as SQL, or MongoDB)
- Messaging/queueing systems (such as RabbitMQ or Service Bus)
- SMTP services for email (such as Postfix)
- Caching systems (such as Memcached or Redis)

Some are locally managed, others are third-party services. The code for a twelve-factor app makes no distinction between local and third-party services. To the app, both are *attached resources*, accessed via a URL or other locator and credentials stored in the config. Different instances of these resources can be swapped out without any changes to the app's code.

5. Build, Release, Run

Strictly separate build and run stages

- The **build stage** is a transform which converts a code repo into an executable bundle known as a build. This might be a single binary, a package, or a container image. The build fetches vendors dependencies and compiles binaries and assets.
- The **release stage** takes the build produced by the build stage and combines it with the deploy's current config. The resulting release is ready for execution…
- The **run stage** runs the app in the execution environment, by launching some set of the app's processes against a selected release.

This means it's impossible to make changes to the *code* or the *config* at runtime, because you can't propagate those changes back to the build or release stage.

Any change to the code creates a new build. Any change to the config creates a new release.

6. Processes

Execute the app as one or more stateless processes

Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.

The twelve-factor app never assumes that anything cached in memory or on disk will be available on a future request or job – with many processes of each type running, chances are high that a future request will be served by a different process.

The memory space or filesystem of the process can be only be used as a brief, single-transaction cache. The twelve-factor app never assumes that anything cached in memory or disk will be available for a future request or job. Even when running only one process, a restart will usually wipe out all local (e.g., memory and filesystem) state.

Sticky sessions are a violation of twelve-factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration, such as Memcached or Redis.

7. Port Binding

Export services via port binding

Twelve-factor apps are completely self-contained, and don't rely on the runtime injections of a webserver to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.

Typically, the web server is a library that's linked using dependency injection, but it happens entirely in user space, within the app code. The contract with the execution environment is binding the port to serve requests.

This approach means that one app can become the backing service for another app, by providing the URL to the backing app as a resource handle in the config for the consuming app

8. Concurrency

Scale out via the process model

In short, the process is the first class citizen. It takes strong cues from the unix process model for running service daemons, assigning each type of work to a process type. Twelve-factor app processes should never "daemonize" themselves, or write PID files. They rely on the operating system's process manager to manage output streams, respond to crashed processes, and handle user-initiated restarts and shutdowns.

The application must be able to span multiple processes running on multiple physical machines.

The share-nothing, horizontally partitionable nature of twelve-factor app processes means that adding more concurrency is a simple and reliable operation.

9. Disposability

Maximize robustness with fast startup and graceful shutdown

The twelve-factor app's processes are disposable, meaning they can be started or stopped at a moment's notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

Processes should strive to **minimize startup time.**

Processes shut down gracefully when they receive a SIGTERM signal.

- For web processes: stop listening, finish any current request, and then exit.
- For worker processes: return the current job to the work queue and then exit.

Processes should also be robust against sudden death. The app must be architected to handle unexpected non-graceful termination. Crash-only design is the logical extreme.

10. Dev/prod parity

Keep development, staging, and production as similar as possible

The old-school (pre-DevOps) way of thinking led to gaps (in time, personnel, and tooling) between development and production. The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.

- Make the time gap small by continuous deployment.
- Make the personnel gap small by following DevOps practices, and involving developers in deployment and monitoring.
- Make the tools gap small by keep development and production as similar as possible.

Resist the urge to use different backing services between development and production. Even tiny incompatibilities between environments create friction that disincentivizes continuous deployment.

In the modern era, declarative provisioning and light-weight virtual environments have brought down the cost and complication of using the same services and tools across all environments.

11. Logs

Treat logs as event streams

Logs are the stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services. They are typically a text format with one event per line, and have no fixed beginning or end, but flow continuously as long as the app is operating.

A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage log files. Each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app's behavior.

In staging or production deploys, each process' stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival.

12. Admin processes

Run admin/management tasks as one-off processes

- Running database migrations
- Running one-time scripts to import, export, or transform data
- Running a console shell to run arbitrary code or inspect the app models against the live database.

One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.

Twelve-factor strongly favors frameworks which provide a REPL shell out of the box, and make it easy to run one-off scripts. In a local deploy, developers invoke one-off admin processes by a direct shell command inside the app's checkout directory. In a production deploy, developers can use ssh or other remote command execution mechanism provided by that deploy's execution environment to run such a process.

Sources

0. This Presentation
1. AKS-Bicep
2. Cluster
3. Agile Manifesto
4. Principles of Agile Software
5. 12 Factor Apps
6. GitOps Principles
7. Control Theory
8. Crash-Only Software
9. Crash-only Software: More than meets the eye
10. History of DevOps (Atlassian)
11. DevOps (Wikipedia)
12. DORA
13. Continuous Delivery (Wikipedia)