DevOps Patterns to Enable Success with Microservices

Richard Mills
DevOps Solution Lead, Coveros Inc.
rich.mills@coveros.com
@armillz
Who is this guy?

- Me: Mad-Software-Developer turned Mad-Software-Engineer turned DevOps-Solution-Architect. Pragmatist. Particular focus on tools and automation. CI, CD, Agile, DevOps ... what’s next?
  - PS: Thanks for inventing the term “DevOps” to describe what I like to do.

- Pays my bills: Coveros helps organizations accelerate the delivery of secure, reliable software using agile methods. We provide expert consulting, mentoring, and training in:
  - Agile transformations, coaching, development,
  - Agile testing and automation
  - DevOps and DevSecOps transformation and implementations
  - DevOps engineering
Why is he here?

• Share some of my experiences (and failures) on a journey from “classic” enterprise monolithic systems into the “utopia” of microservices architecture
• Show some patterns for delivering highly-buzzword-compliant, microservices-based software applications
• Give you a reference to walk away with

• NOT: Explain fundamentals of DevOps (or Agile)
• NOT: Sell you on DevOps (or Agile), necessarily

WARNING: I do everything very fast. Try to keep up.
Agenda and Outline

• Problem - many things changing at different rates (fast vs. never), decoupled interfaces that are hard to verify independently
• Solution - automated pipelines, lots of testing in dynamic test environment
• Pipeline design and patterns
  • General pipeline design
  • Parallel pipelines
  • Test automation stages
  • Simultaneous branch and cross-service testing
  • Simplified versioning to avoid integration problems
  • Centralized Jenkins libraries to avoid duplication
  • Full VM’s vs. streamlined containers
  • On-demand test environments
• Importance of team structure and communication
Microservices: utopia or nightmare?
Microservices are all the rage!

“Let’s take all of our busted monolithic systems and split them up!” they said.

Successful microservices require a combination of technical architecture, automation, and development methodology that all relate closely to Agile and DevOps.

Without the right mix of these, things often don’t go as well as you planned.
Microservices in the non-ideal real world

• Federated services with hidden dependencies
  • Independently changing applications, but ...
  • Loosely coupled, unspecified, AND STILL PRESENT dependencies

• Teams that don’t talk to each other
  • Hide behind (poorly defined) APIs
  • Freedom to move at their own pace

• Desire: Deploy rapidly to production

• Problem: can’t figure out which versions work together, even in test environments

You have many things changing at different rates (fast vs. never) with decoupled interfaces that are hard to verify independently
Signs you DevOps’d your microservices wrong...

• Lack of automation (CI/CD)
• Lack of automated testing
• Constant breaking changes between services
• Manual approvals for deployment
• Monolithic coupling

Each of these can be addressed and avoided with proper DevOps practices – team communication, architecture, and automation
What you need

• Bring teams closer
• Continuously integrate services
• Deploy integrated services into various environments
• Frequently test still-monolithic system in pieces and as a whole

This looks suspiciously like a DevOps problem

Conclusion: your DevOps pipeline is critical to your success
Automation is Critical to Success

“If you are building microservices and you don’t have a highly automated CI/CD pipeline in place, you’ve already lost.”

- some really smart guy I was talking to at some point
DevOps and automation as a solution

• High degree of automation – build, deploy, test, release
• Independent pipelines for individual services
• Global knowledge of system
• Determine if local changes break other services (testing!?)
• Facilitate communication
• Deal with mutually breaking changes

Detect small problems before they hopelessly break the system
Context:
Modern Microservices Example
Dozen-ish independent services acting as a federated product in a registration and workflow system. React/JS front-end UI, Java Spring Boot REST Services, PostgreSQL back-end.

Service Architecture

PostgreSQL RDS instance in AWS
Operating Platform

- Docker containers of all services (application and devops)
- OpenShift Kubernetes Distribution (OKD)
- Amazon Web Services cloud platform
Deployment Architecture

DevOps Cluster

Jenkins | Nexus | Sonar Qube

master | infra | app | app

DEV Cluster

TEST Cluster

PROD Cluster

Amazon Web Services
DevOps Baseline – what do we mean?
The Many Things of DevOps

Notice that not many of these are PURELY the responsibility of one person or even team.
Ok, But What Do We Actually Mean by DevOps?

• DevOps is the natural evolution of Agile: how to get working software into the hands of the people who need it rapidly and reliably
  • Who? Developers, Product Owners, ultimately users
  • When? Now!
    • Seriously ... measure delivery with "minutes"
  • How? Team structure, processes, automation, tools

• Extension of Agile
  • Focus on value delivery
  • Common sense
  • Enable developers to be creative and do great work

• Work with developers to integrate system architecture with deployment architecture
  • Absolutely makes things easier and work better
  • Examples: metrics, testing, health, scaling
A DevOps Solution Includes...

- Team structure and skill set across the delivery chain
- Infrastructure and tools to support automation and team collaboration
- *Delivery pipeline to build, package, deploy, test for repositories and branches of code*
- Test automation (lots and lots of it) capable of evaluating changes

... all wrapped up in a supporting Agile development methodology

Overall: the ability to delivery independent changes into a working environment rapidly and reliability while maintaining a high quality system
Pipeline Design: Patterns of Success
Pipeline defines delivery process

The software delivery process is automated through a CI/CD pipeline to deliver application microservices into various test (and eventually production) environments.
Pipeline links to Branching Strategy

- Strive for "main line" development
  - Use short, small feature branches for isolated changes
  - Consider Github Flow (very simple), Git Flow (complex)
  - Avoid "parallel release development" at all costs
- What problems does branching cause vs. solve?
- Align with your delivery pipeline
Successful branching delivery pattern

• Features – simple CI build with compile, static analysis, unit test
• Pull Request (merge request) – pre-merge with master, deploy to on-demand ephemeral environment, run automated tests (more on tests later)
• Master – after successful merge, build and deploy to dev-staging environment, run more tests, promote
Foreshadowing: branch everything with the code

- Tenet of DevOps and CI/CD: version-control everything so you can automate everything

- We will discuss a lot of different artifacts that match your application code
  - Application code
  - Build scripts
  - Deployment scripts
  - Database scripts
  - Tests
  - ... More

- All of these should be stored and branched as a related code base!

Stand by for station identification...
Pattern: Independent Pipelines

• Each service has its own pipeline (later: shared pipeline code)
  • Example: use “Jenkinsfile” or localized build script configuration

• Major parts of the “independent” pipelines
  • Compile
  • Unit test
  • Static analysis (security, quality, composition analysis)
  • Packaging
  • Integration testing ... Pre/post deployment
  • Test deployment

• Is your service independently deployable?
• How much testing can you do?
• What are your dependencies? (and what will eventually depend on you?)
Pattern: Pipeline should branch with code

• Your build/delivery process will evolve with the code
  • Example: Build scripts ... historically always live with code (maven, gradle, ...)
• Your “pipeline” automation code should live with code, as well.
• Success Pattern: Use “pipeline as code” to define your pipeline automation
  • Store pipeline definition (e.g., “Jenkinsfile”) in repository with code
  • New branches of code can have local build/test pipeline updates

• Similar for deployment and configuration scripts
  • Ansible/Chef – playbooks and recipes change to address code changes (new config parameters, etc.)
  • Docker/Kubernetes/Helm – orchestration and deployment YAML for containers adjust with code

Ideally, these live in the same code repository
Pipelines run in parallel

- Each of your independent service pipelines can (will) execute in parallel
- This can be very resource intensive – plan for it
- At some point, you need to deploy your code into an “environment”
- Will these be shared/mixed environments, or dedicated to the independent pipeline?
- Note: static environment can cause issues if you are changing multiple things simultaneously
- “Dynamic” environments become VERY beneficial...

Service “A”

Service “B”

- OR -
Pattern: Dynamic on-demand environments

- Microservices can have high rates of change across many services
- This can continually disrupt shared/static environments as new services are updates
- The ability to launch a new environment with all supporting services becomes valuable
  - Replicate current PROD (or other) version of services
  - Deploy newly changed service into replica
  - Test, test, test
- *Seeding data becomes important for new copies*
Sidebar: Full VM’s vs. streamlined containers

- Containers make micro-services much easier
- Containers enable easy on-demand test environments
- Containers allow widely disparate application architectures to be deployed on shared infrastructure
- NOTE: You will NEED a container orchestration system (Kubernetes, Swarm, Mesos, Compose, ...)

![Containers vs. VMs diagram](image-url)
Pattern: Database change management

- Microservices frequently need some form of persistence (e.g., database)
- Relational databases often require schema and data changes as the code evolves (non-SQL databases are a different story)
- These changes need to be version controlled and managed just like everything else
- The changes may or may-not have similar change lifecycles to the code
  - Schema
  - Data
- Classes of test data:
  - Reference data (lookup tables, workflow definitions, etc.)
  - Sample data (basic data to show functionality)
  - Test data (more complicated data to support testing)

- RDBMS tool examples: Liquibase, Flyway, ...

All of this can (should) be stored and branched with the code
Pipeline “join” points

• At some points, your pipelines come together and services work in a combined environment
• You will need to deploy multiple services and make them work together
• Need some capture of “versions” to know which versions are where and how they are verified against each other
• These are typically extensions of independent pipelines
• MIGHT be ”join” jobs that deploy lots of things together
• Consider “locking” environment during testing to avoid blips and test failures
Side Bar: Versioning ... do you need it?

- Is your app a multi-version beast, or simple end-of-the line single deployment?
- Keep it simple, (stupid)
- Semantic versioning vs. unique identifier
  - Integrate version control identifiers with deployed code
- All software elements need a traceable identifier
  - Source Code -> Deployable Package -> Running Software
- Very few deployable applications need linear versioning
  - Does it matter if it’s 1.2.3 or 20180317 or a3e78b19d?
  - Consider using git SHA version label to enable easy traceability
- Component libraries with APIs frequently need identifiable, increasing versions
  - my-util:1.2.3 vs. 1.3.9 with different capabilities that link to your code interfaces
Capturing a product version

• As your services progress through environments, it’s useful to know what versions of services work with each other

• Your “promotion” mechanism might need to consider moving groups of services together
  • Single service
  • Group of known-good services
  • This may identify your subsystem of coupled monoliths

• Capture versions of your services
  • SHA, semver, whatever
  • Typically capture in single “release inventory” file to feed to deployment automation scripts
Topic: UI and REST service code integration

• Problem: UI invokes back-end services through APIs
• What happens when they need to change?
• How do you test these continuously?

• Scenario 1: Full-stack developers building UI and API
  • Consider having UI and API code live together
  • Aligns UI and API changes automatically

• Scenario 2: Separate UI and API teams
  • UI and API might change at different rates
  • Need mechanism to alter/change APIs (deprecate, multiple versions, etc.)
  • UI and API might be in different repositories
Deployment automation and configuration

- Installation and configuration automated through CM automation framework
  - Docker, Kubernetes, Ansible, Chef
- Example:
  - deployment of microservices is automated and version controlled through OpenShift templates
  - Secrets/properties injected from OpenShift
- Unique environment configuration specified in property configuration files managed by deployment pipeline
  - staging-staging.properties
  - uat-uat.properties
Ensuring quality (i.e., testing)
Your efforts in DevOps (and microservices) will fail without proper automated testing and assessment that is fully integrated into the pipeline.
Goals for Testing in DevOps

• Keep software in continuous working state
• Establish confidence in change
• Force teams to build quality in (and agree it’s important)
• Avoid creating too much technical debt too early

Important: everywhere I say “Quality” I mean “Quality, Security, Performance, and all the other –ilities” you can think of for your software.
Test factors to consider with microservices

• Take every opportunity you can to test the code
• This will enable rapid changes later

Testing paths to consider
• Testing in isolation (totally in your control) – verify what you think your contract is
• Testing with downstream dependencies (database, other services)
  • What else do you need to work properly?
  • Verify what you think other contracts are
• Testing with upstream dependencies (UI, other services)
  • Who needs YOU to work properly?
  • Verify what other people think your contract is

• ”THE” pipeline needs enough global knowledge to orchestrate this

• Pathological case: what happens when APIs change unexpectedly or don’t fulfill their (poorly specified) contracts?

• PICTURE: Multiple services that need each other
Test stages in the pipeline
Tests live with code and pipeline

• Tests for microservices should live with the microservice code
• Unit tests ... Part of code tree (e.g., junit)
• Automated REST tests ... Part of code repository (Postman, REST Assured, ...)
• UI tests ... Part of UI repository

• This allows you to easily match code version to test version

*Pro tip: bundling tests in a docker container makes them easy for anyone to execute*
Challenge: cross-service and cross-branch testing

• Major problem: multiple services (e.g., UI or multiple APIs) need similar linked changes
• How do I implement this?
  • Code based solutions (feature flags, API versions, ...)
  • Deployment based solutions (concurrent running service versions)
• More important: how do I test this?
  • Will depend on your implementation
  • May require the ability to link changes between services/repositories
• Simultaneous branch and cross-service testing
  • Example: specify multiple builds or deployment versions during test environment creation
• Anti-pattern: multiple versions of all APIs deployed everywhere
  • How do you wire them together?
• Simplified versioning to avoid integration problems
  • Force the integration problems as soon as possible
Coding Your Pipeline
Individual pipeline implementation

• CD/CI delivery pipeline as code
• Each microservice implements a very simple “Jenkinsfile” build script which uses central pipeline libraries implemented by the DevOps support team
• Simple is good ... avoid massive replication

```plaintext
core_pipeline {
    continuous_integration(ephemInfo)
    on_pull_request {
        continuous_delivery_pr(ephemInfo)
    }
    on_merge_to_master {
        continuous_delivery_master()
    }
}
```
Pipeline as code structure

- Jenkins 2.x - Pipeline as Code (gone: mega multi-job structures)
- “Github Organization” Jenkins job scans all repositories
- Each code repository has Jenkinsfile that invokes core libraries
- Pipeline DSL shared libraries
  - Jenkins jobs – all captured as version controlled DSL, can be customized per service
  - Core Pipeline Stages - build/deploy stages (overridable)
  - Core Pipeline Libraries – shared utilities (locked)
Centralized Jenkins libraries

- Pipelines for similar services will naturally look very similar
- Centralize the bulk of the shared activities into code libraries
- Customizable for unique independent team needs
- Team note: you may need a central team to maintain and coordinate this
- Branching
  - Need to branch/test this code just like any other
  - Also need ways for independent pipelines to use new or old versions
Important Non-Pipeline Aspects
Importance of team structure and communication

• Teams must align with delivery process
• Your teams are likely part of a shared product
  • If not ... Great!
  • If so, don’t pretend to be separate

• Communicate with each other

• Regardless, you will likely have shared patterns (devops, testing, architecture) that can be leveraged to avoid re-inventing the wheel
Example “DevOps” product team structure

Our team organization on a 50-ish person development project

Important for the vertical development teams to actually work with each other (painful at first)
Integrating with Software Architecture

- Not all software lends itself to easy build, test, and deployment
- Work directly with software architecture and development teams
- Software must support
  - Rapid build
  - Automated test (controllability, observability)
  - Data initialization
  - Installation/configuration
  - Monitoring/metrics
- Standards and "Definition of Done" should reflect this
- Development stories aren’t complete until all these things work
Take-aways

• Version control everything, automate everything
• Your pipeline implements your branching strategy
• Independent pipelines with join points
• Use dynamic environments (containers!)
• Store and branch everything together – code, tests, database, pipeline code
• Test in isolation, test with downstream, and test with upstream
• Use shared/central pipeline libraries to avoid duplication and enforce standards

• And don’t forget:

   WORK TOGETHER, DAMMIT!
Questions?

Thank you!

rich.mills@coveros.com

https://www.coveros.com/services/devops/