DevOps and Cloud Native Development

IBM Cloud

IBM Point of View: DevOps and Cloud Native Development

DevOps is a fundamental part of our opinionated modern day delivery methodology…

The cloud is the modern platform for modern architectures and applications…

IBM Cloud Garage Method

Combining industry best practices for **Design Thinking, Lean Startup, Agile Development, DevOps,** and **Cloud** to build and deliver innovative solutions.

Modern delivery practices , behaviors, and architectures for modern applications

A holistic approach to modern delivery

Architecture Center

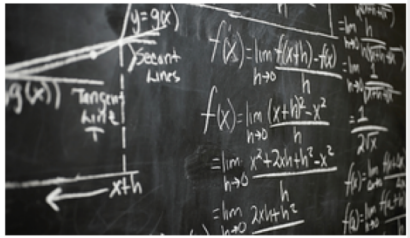
Cloud architectures provide specific technology, practice and tool choices to build and deploy world class enterprise applications.

Garage Method Practices

Combine industry practices including IBM Design Thinking, Lean Startup, agile development, and continuous delivery to build innovative solutions.

DevOps Toolchains

Combine IBM Cloud services with open source and third-party tools to enable your team as you adopt Garage methodology.



Courses and tutorials

Learn concepts of the Garage Method and test your knowledge.

Get technical →

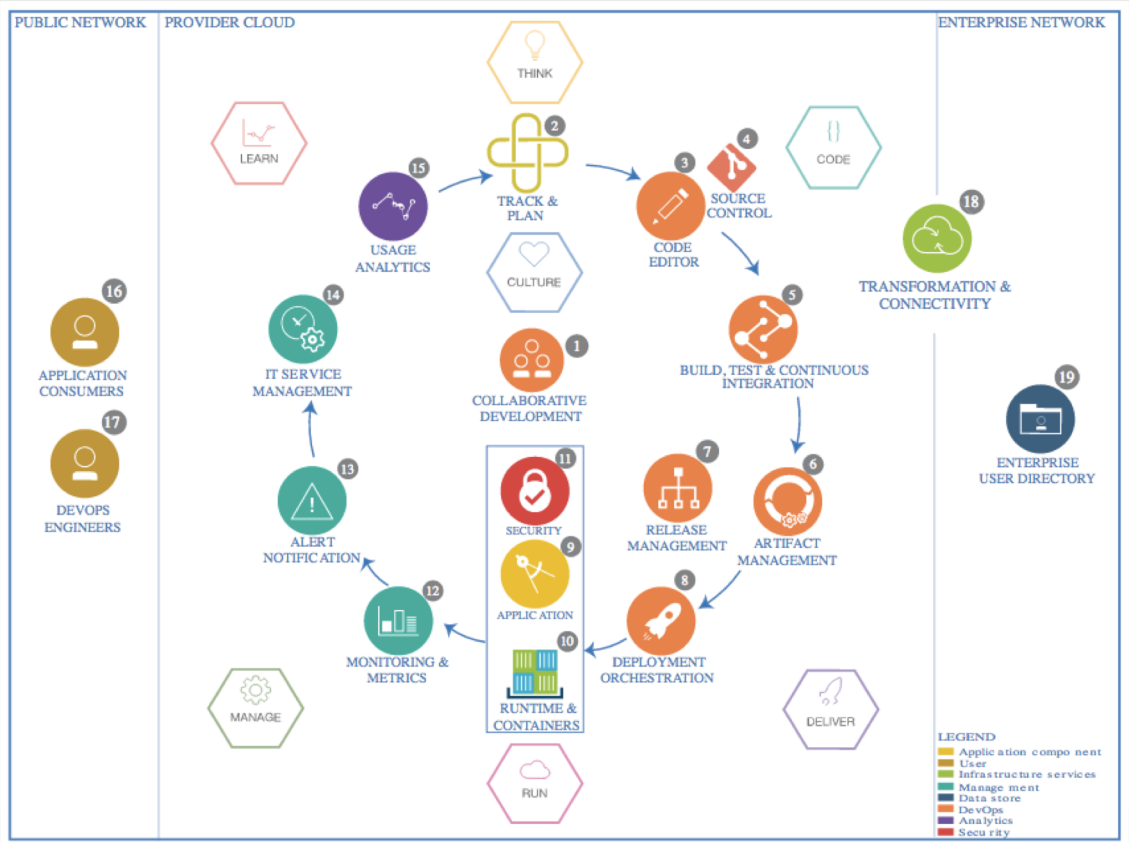
Adopt the practices →

Start developing →

Toolchains can be hosted on any cloud, and deploy to any cloud (including IBM Cloud Private)

Build skills →

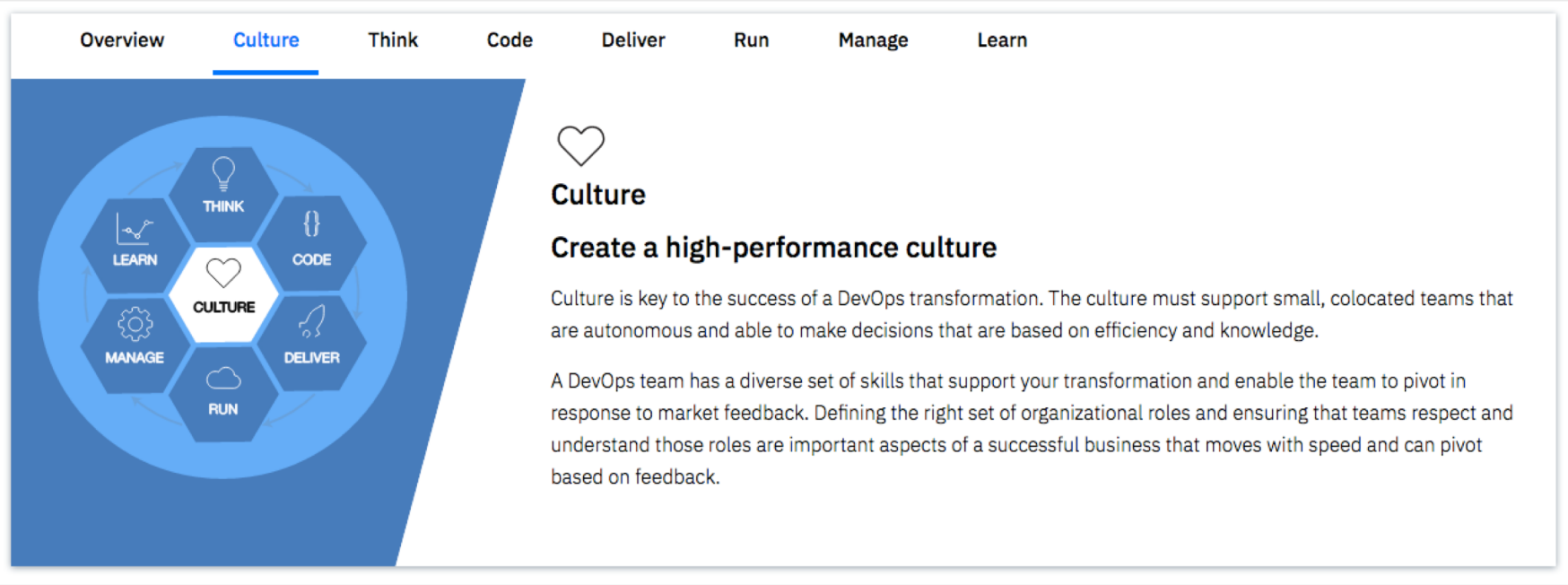
IBM DevOps Reference Architecture



IBM's opinionated architecture that can address delivery of cloud native, cloud ready and hybrid traditional applications to public, private, and hybrid clouds.

https://www.ibm.com/cloud/garage/architectures/devOpsArchitecture/reference-architecture

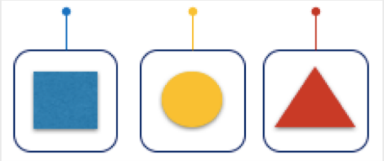
Culture…the lynchpin of success



- Team structures must change.
- New roles must be formed.
- New behaviors must be learned and practiced.
- Liberties must be ceded back to teams…to unleash them.

Modern platforms,
Modern architectural styles

Cloud Native development



Microservices and APIs are the modern day architectural style, which leverage cloud platforms

The availability of cloud platforms, virtualization advancements, and the emergence of Agile and DevOps practices has thoroughly upended traditional software development practices. Not only that, but traditional monolithic applications (including their supporting infrastructure) were not built to take advantage of flexible cloud resources.

Microservices, as an architecture, describes an approach to building complex applications using independent, replaceable processes. These processes are single-purpose, with a small scope. They communicate with other services using lightweight APIs and language-agnostic protocols. These independent processes are also called microservices.

Isolating each piece of an application can reduce the overhead involved in making changes and can reduce the risk around trying something new. But this approach has a down side: distributed systems are not a trivial problem space to build in.

https://developer.ibm.com/cloud-microservices/

Core ingredients

There is no magic around microservices. These tidy little services can become a spaghetti mess just like software built with any other approach. **Thinking and team organization need to change to make microservices successful**.

Microservices are independent. Agility is one of the benefits of microservice architectures, but it only exists when services are capable of being completely re-written without disturbing other services. That isn't likely to happen often, but it explains the requirement. Clear API boundaries give the team working on a service the most flexibility to evolve the implementation. This characteristic is what enables polyglot programming and persistence.

Microservices are resilient. Overall application stability depends on individual microservices being robust to failure. This is a big difference from traditional architectures, where the supporting infrastructure handled failures for you. Each service needs to apply isolation patterns like circuit breakers and bulkheads to contain downstream failures and define appropriate fallback behaviors to protect upstream services.

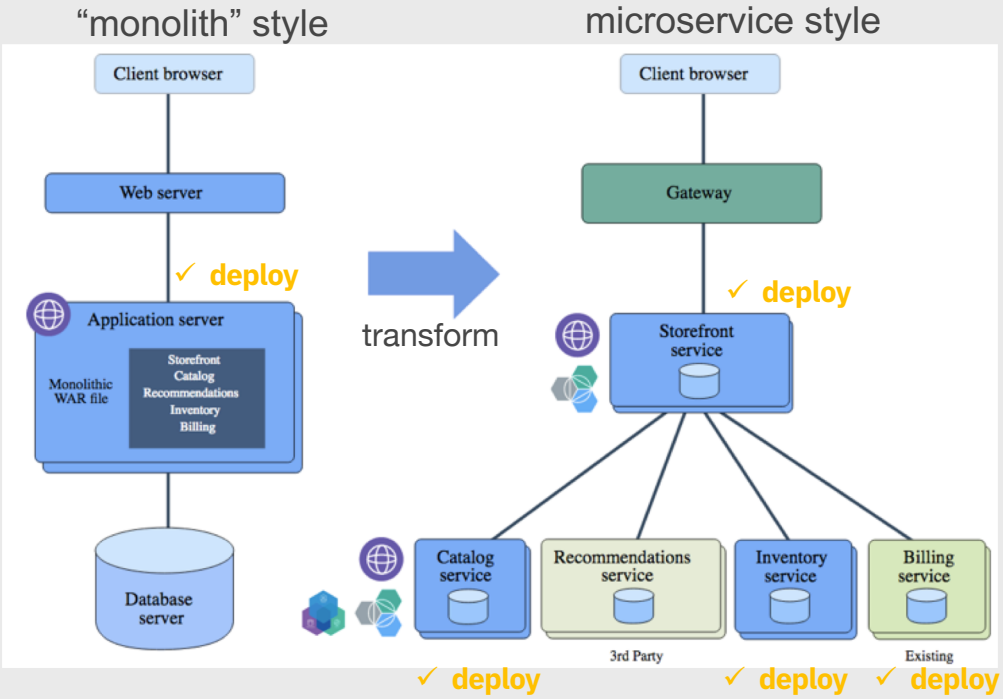
Microservices are stateless, transient processes. This is not the same as saying microservices can't have state! It does mean that state should be stored in external backing cloud services, like Redis, rather than in-memory. Fast startup and graceful shutdown behavior further allows microservices to work well with automated orchestrators that create and destroy instances in response to load.

THE BREAKDOWN

Why is DevOps essential?

By their very nature, microservices will have more deployments, due to the increase in componentization and change volume.

Without automation in the delivery lifecycle, microservices can quickly become an unmanageable IT endeavor.



Controlling chaos (with DevOps)

By definition, applications that use microservices have many moving parts. Automated tools make the difference between a flexible, scalable, and yet stable system, and an unmanageable mess.

Continuous integration and deployment (CI/CD) pipelines are the base upon which a microservices application is built. Unique pipelines per service ensures that services can evolve at their own pace. The ideal is for any given built instance to transition through stages of a pipeline for eventual deployment into production without requiring any code changes along the way.

Containers help enable CI/CD, as they naturally define the entire operating environment for the service. Using containers assists in eliminating hard-to-spot differences between development and production environments, which are a common source of last-minute deployment issues.

Different services will have different scaling requirements: for example, a catalog service on black Friday will use resources very differently than an account service. CI/CD pipelines perform a role in defining per-service scaling policies as they automate deployment into different environments. Orchestration can then use those policies to manage routing and load balancing accordingly.

Culture shock

There is a huge cultural component to microservices. Conway's law is often cited for a valid reason: **organizational structure does influence how software is built.**

As an example, if an organization is divided so that application developers are separate from database administrators, the resulting architecture will have distinct tiers: a tier optimized for application code, and a data tier that is optimal for database administrators. Application changes that require database changes have to then be coordinated between the two distinct groups. That should sound familiar.

To build truly independent microservices, each service needs to be owned, maintained, and operated by a single team. That team owns the details of how the service works. There will be some coordination required with other teams, but that coordination should be at the surface level. Clearly defined, versioned APIs that use language-agnostic protocols help keep coordination required between teams focused on the important external characteristics of the service, and away from the internal details, which should remain free to change at any time.

Release strategy

Releases are comprised of sets of loosely coupled microservices, delivered by multi-disciplinary squads, leveraging **automation, virtualization, instrumentation, pervasive testing**, and **a high degree of collaboration**

Successful release strategies employ guiding principles:

- **Culture** of Agile, DevOps, Design Thinking, Lean
- Multi-disciplinary **Squads**, responsible for lifecycle delivery (Developers code, test, deploy, debug, maintain)
- **12 Factor App** – tenants that apply specifically to DevOps and delivery
  - Codebase – one code repo/microservice
  - Build, Release, Run — a discrete separation of each phase in delivery
  - Disposability – capability to tear down and spin up microservices simply and quickly
  - Dev/Prod Parity – minimization of variance between development and production environments
- One **toolchain** instantiation per microservice
- **Bounded business context** with well-formed APIs (contracts)
- **Multi-speed IT** enablement (service virtualization, API testing; Test early, test often)
- **Blue-Green Deployment, Dark Launches, Feature Toggles**

IBM Cloud Garage Method

THE TWELVE-FACTOR APP

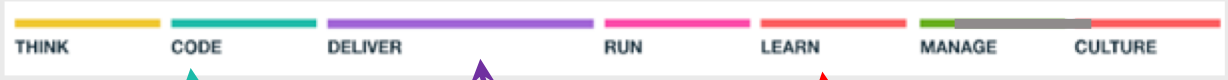
- **Codebase**
- Dependencies
- Config
- Backing Services
- **Build, release, run**
- Processes
- Port binding
- Concurrency
- **Disposability**
- **Dev/prod parity**
- Logs
- Admin Processes

Code quality monitoring and management

Without consideration for code quality, service monitoring, or management, iterative continuous delivery of microservices can lead to:

- more defects
- delayed integration
- slower end-to-end delivery
- lower quality
- more risk

Mitigate risk in delivery with good microservice design practices (12 Factor App), continuous static and dynamic testing, and a culture of continuous feedback and optimization.



Implement observability patterns

Enable granular and dynamic monitoring by designing microservices

Test source quality

Infuse static analysis tools in the pipeline to scour source code to encourage implementation of leading practices (12 Factor App).

Test code resiliency

“Force Fail” runtime microservices, to observe behavior on failure conditions.

Improve code quality

Refactor microservice to address defects in design, source code, and behavior.

Deployment patterns and service management

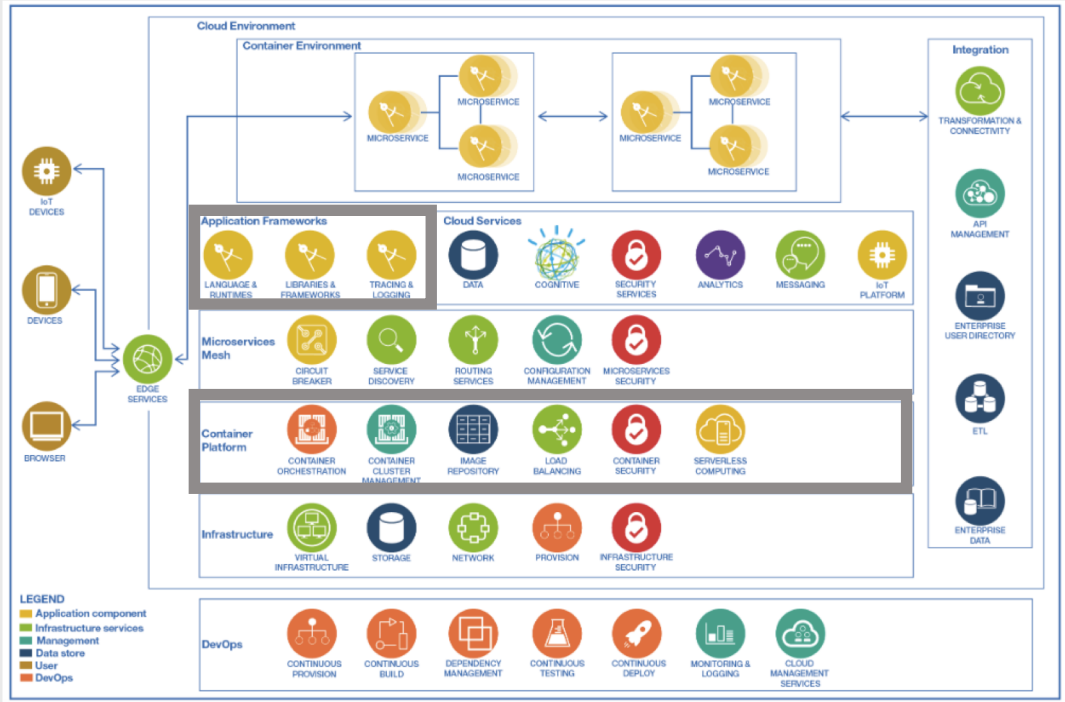
Microservices outnumber their monolith equivalents

Best hosted on elastic, commoditized, disposable environments driven by configurable policies and event-driven conditions.

Enabling technologies include:
- Docker and Cloud Foundry
- IBM Cloud Functions and AWS Lambda

Microservices should:
- adhere to 12 Factor App tenants
- exploit a service mesh
- exploit coding frameworks

Enabling technologies include:
- Istio
- Netflix OSS
- Spring Boot



IBM Microservices Reference Architecture

Service Management Considerations for Microservices:
- Leverage circuit-breaker pattern
- Exploit “observability” patterns
- Leverage auto-scaling policies
- Routinely deploy chaos testing
- Use “Blue Green” deployments

Responsive operations

Continuous feedback, learning, and optimization is critical in landscape with more moving parts (microservices)

In today's global marketplace, websites are expected to be always available.

1 **Optimize delivery with a DevOps dashboard**

A DevOps Dashboard tracks and measures velocity and quality of release trains (Design through Deploy)

2 **Enrich user experiences with runtime analysis**

Inspect runtime dynamics of microservices to understand memory, I/O, workload, runtime container behavior. Based on runtime behavior, tune/refactor microservice.

3 **Harden with chaos testing and runbooks**

Build a culture of reliability with chaos testing.

4 **Make visibility and accountability easy**

Operations is the responsibility of all squad members.