Application Modernization

Objectives

Key objectives of this chapter

- What is Application Modernization?
- Typical App Modernization Projects
- Why Modernize?
- Goals for Modernization
- Twelve-factor Application Microservices
- Maintaining State
- Cloud Service Fabric

1.1 What is Application Modernization

- Application modernization addresses the migration of existing business critical applications (AKA legacy) to new applications, platforms and component-based models.
- Application modernization includes the integration of new functionality to provide the latest functions to the business.
- Modernization options include re-platforming, re-hosting, recoding, re-architecting, re-engineering, interoperability, replacement and retirement.
- Modernization may include changes to the application architecture to clarify which option should be selected.

1.2 Typical App Modernization Projects

- Re-hosting legacy applications by shifting applications from a mainframe, e.g. zSeries, or midframe, e.g. iSeries, environment to Windows or UNIX in order to reduce maintenance, hardware, and/or software costs.
- Wrapping legacy applications with Service-oriented architecture (SOA) or microservices interfaces.
- Automated migration of legacy data and applications; transforming application code developed in legacy platforms to Java or .NET
- COTS (Commercial off-the-shelf); replacing legacy applications with new packaged applications such as SAP, PeopleSoft and Seibel.
- Re-architecting legacy applications; extracting the business assets from legacy applications and transforming them on to a modern architecture using modern tools and development platforms

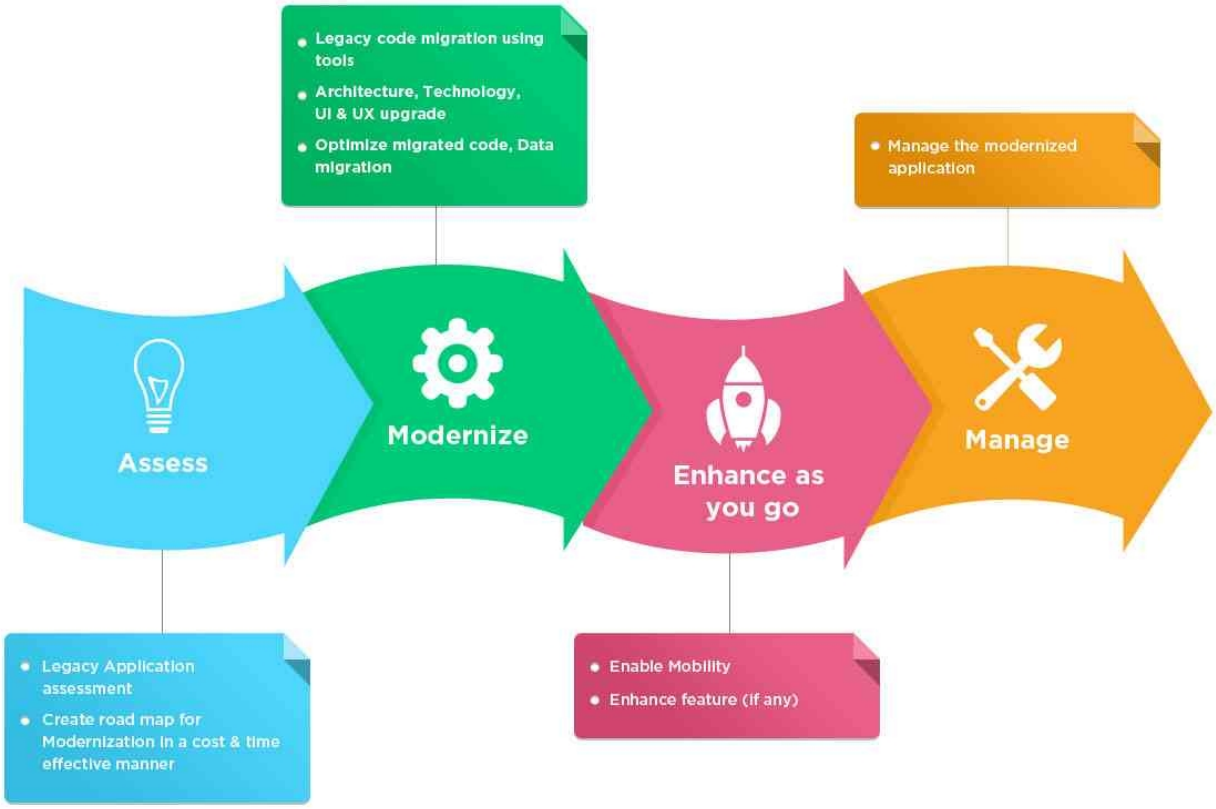
1.3 Why Modernization?

- Where does the business need to be in six months? Two years? Five years?
- What are the major challenges users of the applications have today?
- What risks does the current business critical technology present to the business?
- What are the challenges of supporting current applications?
- What technologies and architecture are we using for green field software?
- What costs of maintaining our environments can be reduced by modernization?
- Are there any revenue increases that can be achieved by improving, adding or changing technology and processes?

1.4 Goals for Application Modernization

- Business value – a primary goal of an application modernization project is to create new business value from existing applications.
- Agility – many existing systems are difficult to modify or add features. By considering areas like Minimum Viable Product (MVP) and time-to-market, we wire agility into the design and architecture of our modernized applications
- Revenue - keeping complex business critical applications running smoothly can be a costly, time-consuming, resource-intensive process, especially when the software becomes outdated or incompatible with newer versions of the underlying operating system (OS) or system hardware.
- Flexibility - in many existing applications, the business process workflow is hardcoded and tightly coupled with other aspects of the legacy code. Application modernization should consider the flexibility of loosely coupled application components and services

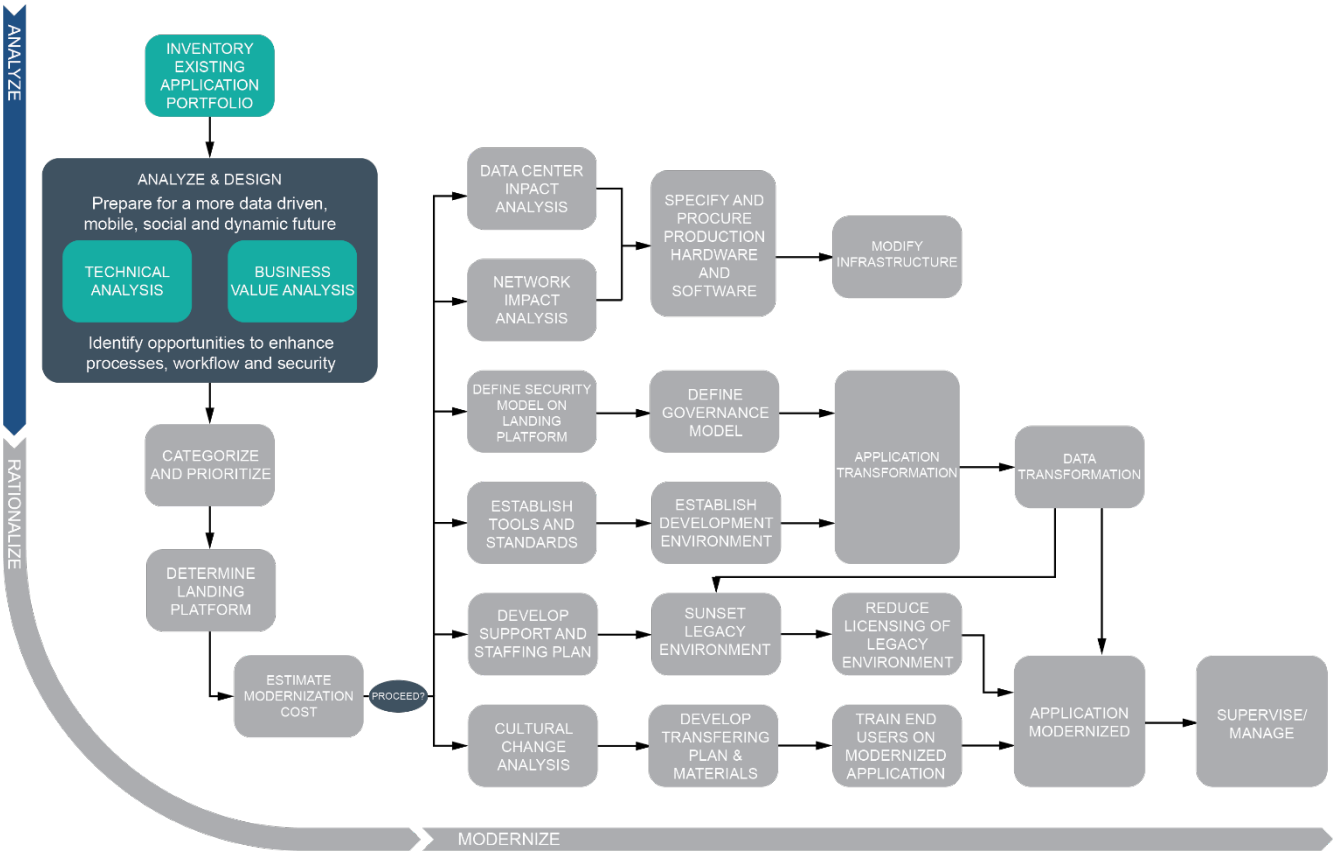
1.5 Modernization Process



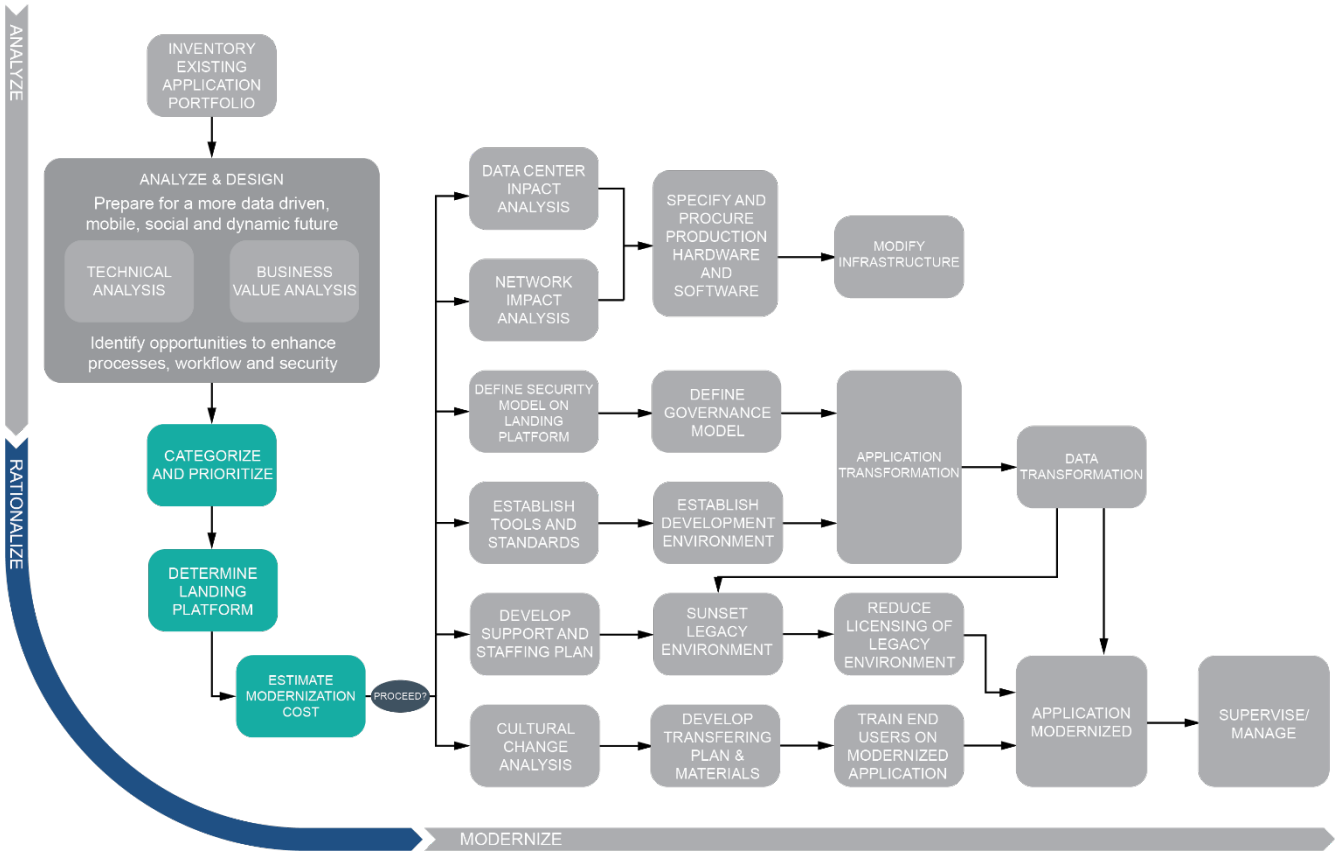
1.6 Modernization in a Nutshell

- Analyze – understand what applications are priority for modernization and how we keep the existing running in parallel with the new application
- Rationalize – why is this application key to the current modernization effort and how is that change going to drive revenue or customer engagement
- Modernize – move the application to the future state architecture, platform and software
- Supervise – manage the applications being modernized from the onboarding funnel, ongoing modernization, and in the usage by customers

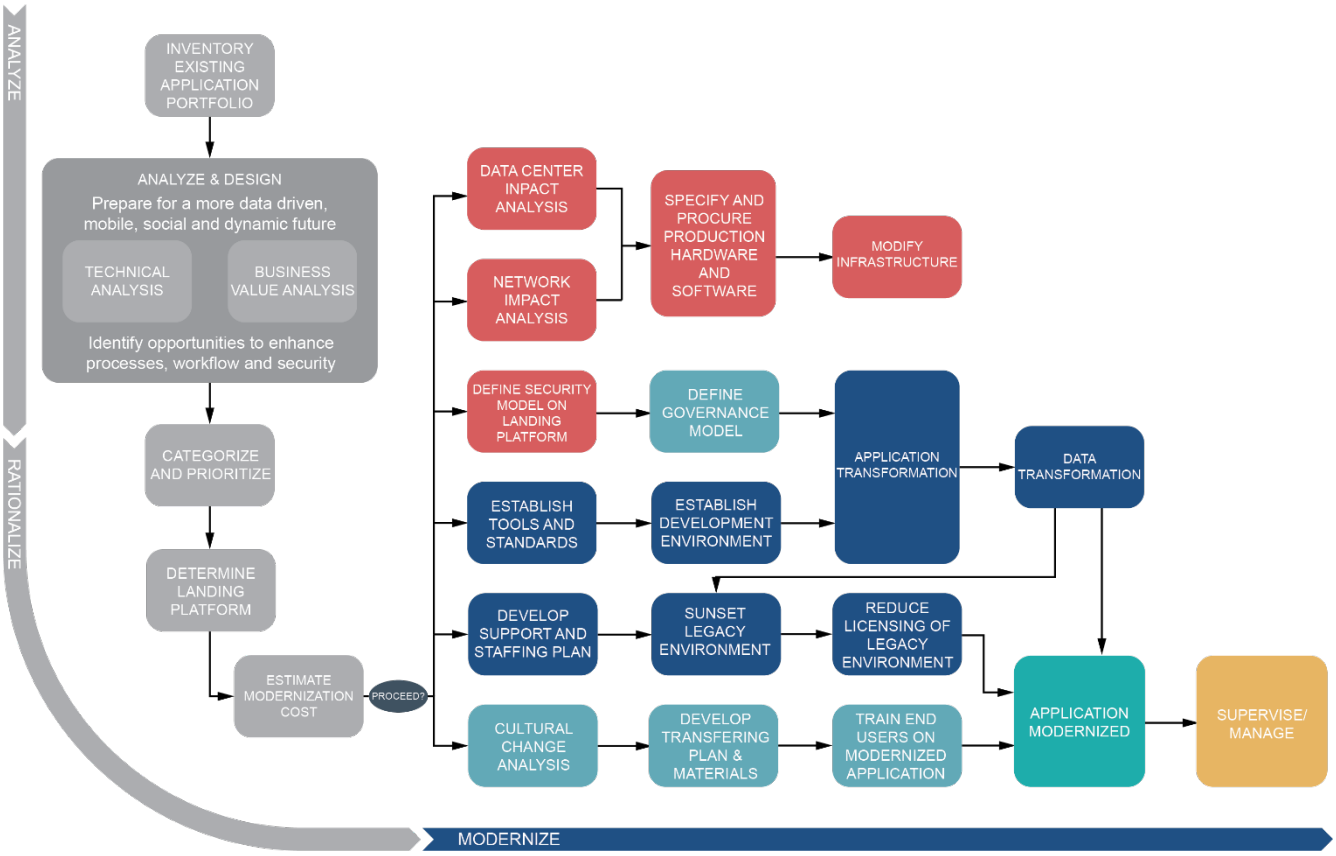
1.7 Modernization in a Nutshell - Analyze



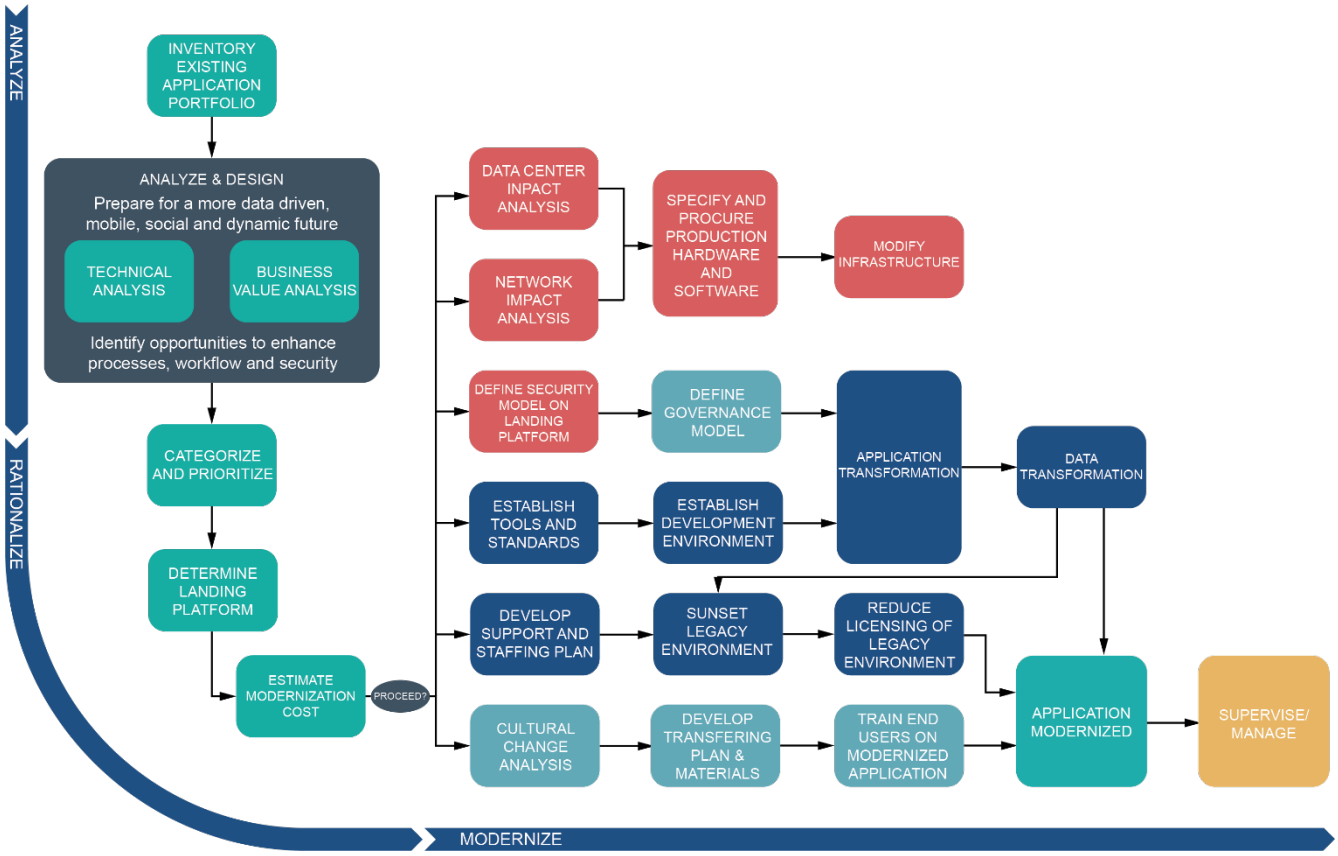
1.8 Modernization in a Nutshell - Rationalize



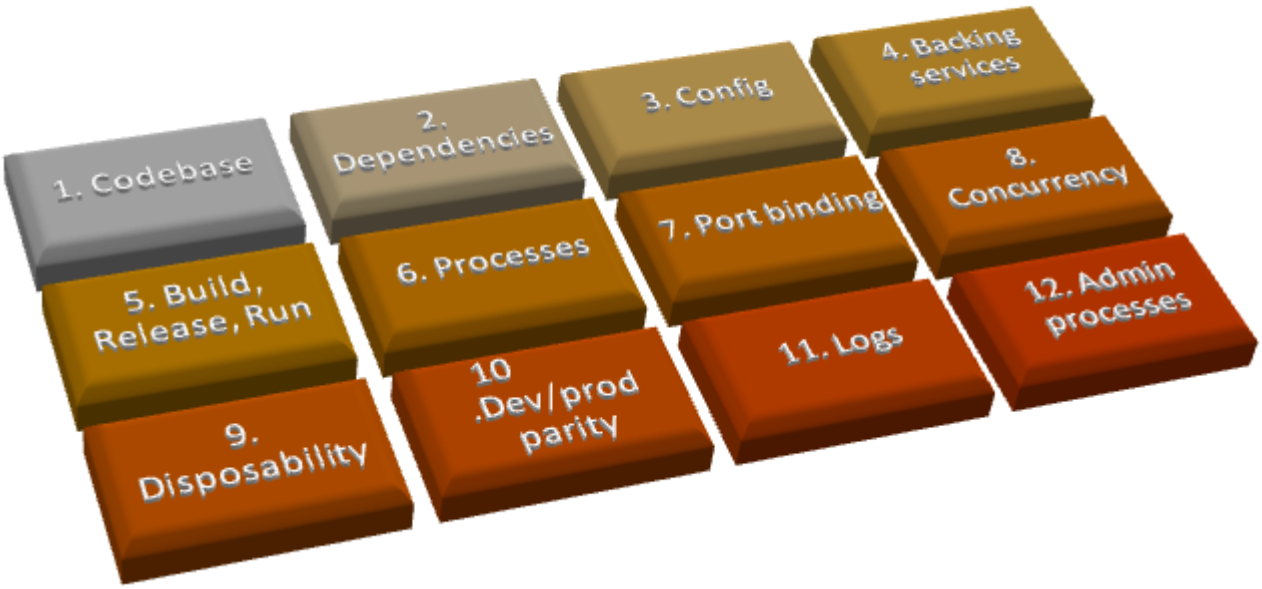
1.9 Modernization in a Nutshell - Modernize



1.10 Modernization in a Nutshell – Supervise



1.11 Twelve-factor Applications



1.12 Twelve Factors, Microservices, and App Modernization

- Heroku, a platform as a service (PaaS) provider, established general principles for creating useful web apps known as the Twelve Factor Application
- Applying 12-factor to microservices requires modification of the original PaaS definitions
- Goal of combining microservices, twelve-factor app and app modernization is a general purpose reference architecture enabling continuous delivery

1.13 12-Factor Microservice Codebase

- One codebase per service, tracked in revision control; many deploys
- The Twelve-Factor App recommends one codebase per app. In a microservices architecture, the correct approach is one codebase per service.
- This codebase should be in version control, either distributed, e.g. git, or centralized, e.g. SVN.

1.14 12-Factor Microservice Dependencies

- Explicitly declare and isolate dependencies
- As suggested in The Twelve-Factor App, regardless of what platform your application is running on, use the dependency manager included with your language or framework.
- Do not assume that the tool, library or application your code depends on will be there.
- How you install operating system or platform dependencies depends on the platform:
- In noncontainerized environments, use a configuration management tool (Chef, Puppet, Salt, Ansible) to install system dependencies.
- In a containerized environment, do this in the Dockerfile.

1.15 12-Factor Microservice Config

- Store configuration in the environment
- Anything that varies between deployments can be considered configuration.
- All configuration data should be stored in a separate place from the code, and read in by the code at runtime, e.g. when you deploy code to an environment, you copy the correct configuration files into the codebase at that time.
- The Twelve-Factor App guidelines recommend storing all configuration in the environment, rather than committing it to the source code repository.
  - Use non version controlled .env files for local development. Docker supports the loading of these files at runtime.
  - Keep all .env files in a secure storage system, such as Hashicorp Vault, to keep the files available to the development teams, but not committed to Git.
  - Use an environment variable for anything that can change at runtime, and for any secrets that should not be committed to the shared repository.
  - Once you have deployed your application to a delivery platform, use the delivery platform's mechanism for managing environment variables

1.16 12-Factor Microservice Backing Services

- Treat backing services as attached resources

- The Twelve-Factor App guidelines define a backing service as “any service the app consumes over the network as part of its normal operation.”
- Anything external to a service is treated as an attached resource, including other services. This ensures that every service is completely portable and loosely coupled to the other resources in the system.
- Strict separation increases flexibility during development – developers only need to run the service(s) they are modifying, not others.
- Database, cache, queueing system, etc. These should all be referenced by a simple endpoint (URL) and credentials, if necessary.

1.17 12-Factor Microservice Continuous Delivery

- Strictly separate build and run stages

- To support strict separation of build, release, and run stages, as recommended by The Twelve-Factor App, use a continuous integration/continuous delivery (CI/CD) tool to automate builds.
- Docker images make it easy to separate the build and run stages. Ideally, images are created from every commit and treated as deployment artifacts.

1.18 12-Factor Microservice Processes

- Execute the app in one or more stateless processes

- For microservices, the application needs to be stateless.
- Stateless services scale a service horizontally by simply adding more instances of that service.
- Store any stateful data, or data that needs to be shared between instances, in a backing service

1.19 12-Factor Microservice Data Isolation

- Each service manages its own data

- Allow access to the persistent data owned by a service only via the service's API.
- Do not use implicit service contracts between microservices by sharing data between services.
- Service data separation or isolation ensures that microservices do not become tightly coupled.
- Data isolation allows the developer to choose, for each service, the type of data store that best suits its needs, e.g. RDBMS, NoSQL, ...

1.20 12-Factor Microservice Concurrency

- Scale out via the process model

- The Unix and Mainframe process models are predecessors to a true microservices architecture, allowing specialization and resource sharing for different tasks within a monolithic application.
- For microservices architecture, we horizontally scale each service independently, to the extent supported by the underlying infrastructure.
- Docker or other containerized services, provide service concurrency.

1.21 12-Factor Microservice Disposability

- Maximize robustness with fast startup and graceful shutdown

- Instances of a service need to be disposable so they can be started, stopped, and redeployed quickly, and with no loss of data.
- Services deployed in Docker containers satisfy this requirement automatically, as it's an inherent feature of containers that they can be stopped and started instantly.
- Storing state or session data in queues or other backing services ensures that a request is handled seamlessly in the event of a container crash.
- Backing stores support crash□only design.

1.22 12-Factor Microservice Environment Parity

- Keep development, staging, and production as similar as possible

- Keep all of your environments – development, staging, production, and so on – as identical as possible, to reduce the risk that bugs show up only in some environments.
- Containers enable you to run exactly the same execution environment all the way from local development through production.
- Differences in the underlying data can still result in runtime changes in application behavior

1.23 12-Factor Microservice Logs

- Treat logs as event streams

- Use a log management solutions in a microservice for routing or storing logs.
- Define logging strategy as part of the architecture standards, so all services generate logs in the similar fashion

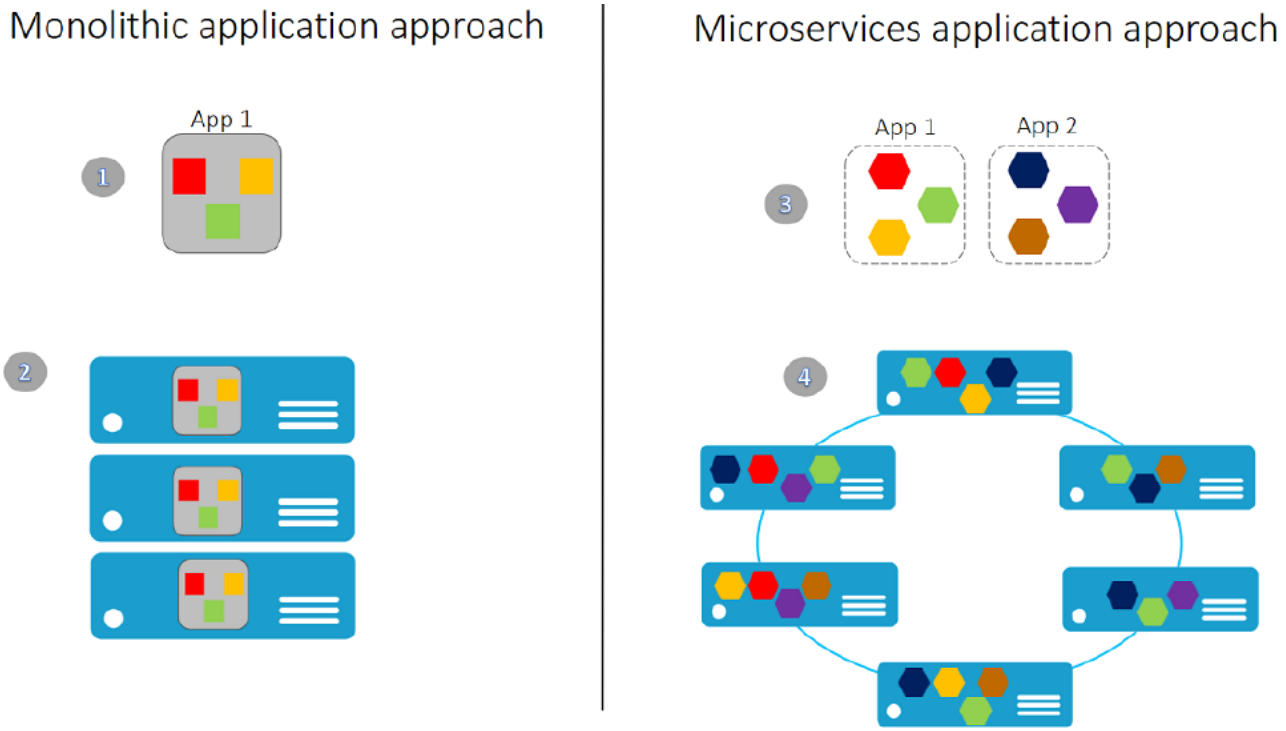
- Log strategy should be part of a larger Application Performance Management (APM) or Digital Performance Management (DPM) solution tied to the Everything as a Service model (XaaS)

1.24 12-Factor Microservice Admin Processes

- Run admin and management tasks as one-off processes

- In a production environment, run administrative and maintenance tasks separately from the app.
- Containers make this very easy, as you can spin up a container just to run a task and then shut it down.
- Examples include, doing data cleanup, running analytics for a presentation, or turning on and off features for A/B testing.

1.25 Monolithic revisited

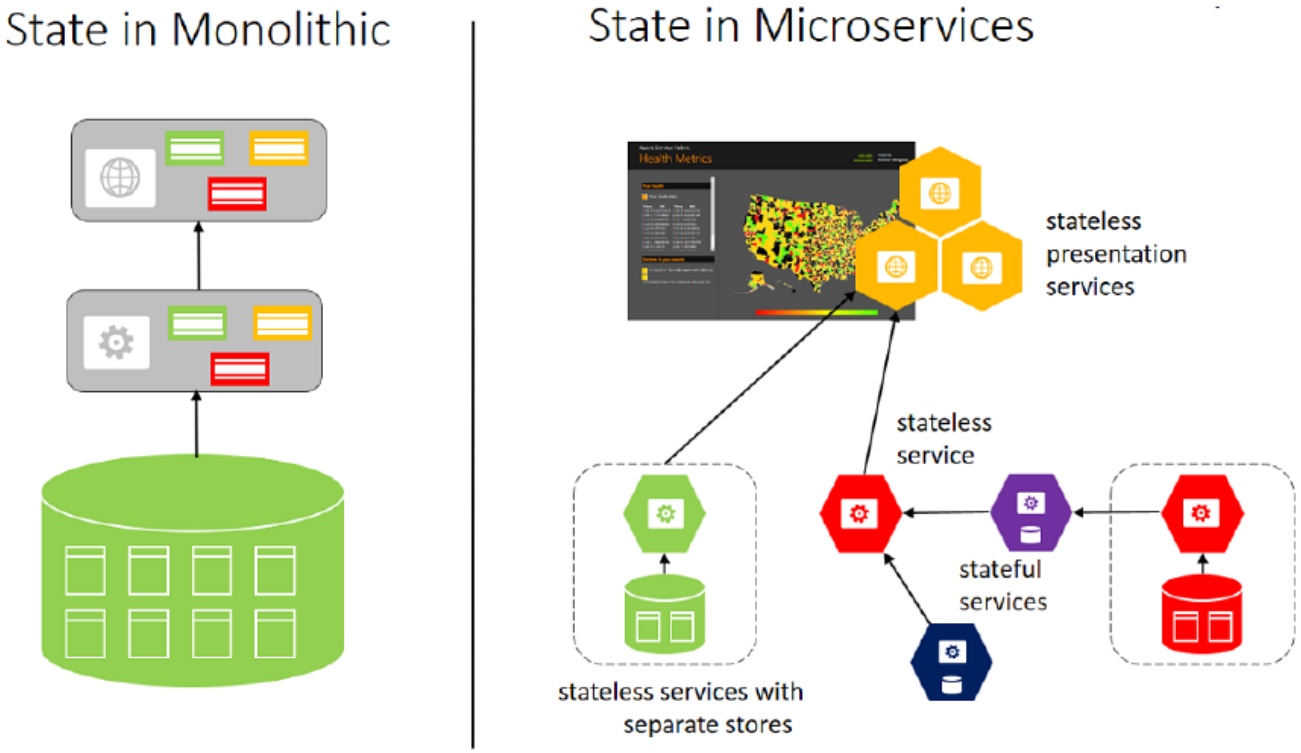


1.26 Monolithic vs. Microservices

1. A monolithic app contains domain-specific functionality and is normally divided by functional layers, such as web, business, and data.
2. Scale a monolithic app by cloning it on multiple servers/virtual machines/containers.
3. Microservice applications separates functionality into separate smaller services.
4. Microservices approach scales out by deploying each service independently, creating instances of these services across servers/virtual machines/containers.

- The microservice architectural style is an approach to developing a single application as a suite of small services.
- Each service runs in its own process and communicates with lightweight mechanisms, often an HTTP resource API
- These services are built around business capabilities and independently deployable by fully automated deployment machinery in DevOps fashion

1.27 Maintaining State in App Modernization



1.28 Cloud Service Fabric

- Cloud providers implement system services to deploy, upgrade, detect, and restart failed services, discover services, route messages, manage state, and monitor health.
- Cloud services provide this service fabric to enable the characteristics of microservices including:
- Ability to deploy applications either running in containers or as processes.
- APIs, to allow programmatic, manual, or orchestrated interaction with
  - secrets/configuration/storage management
  - health checks, auto-[scaling/restart/healing] of containers and nodes
  - zero-downtime deploys

1.29 Summary

- Application modernization can be as straight forward as a lift-and-shift or a complete rewrite of the software, architecture and platform for an application
- Microservices, based on SOA, provide a model for componentized application modernization
- Modernization requires the team to understand the business value and processes of the existing application to maximize the effective of the changes to the application
- Application modernization expects teams to leverage technologies and services like log management, cloud, continuous integration and delivery
- DevOps is a core piece of application modernization