**Key Design Principles**

When getting started, keep in mind the key principles that will help you to create an architecture that adheres to proven principles, minimizes costs and maintenance requirements, and promotes usability and extendibility. The key principles are:

* **Separation of concerns**. Divide your application into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling. However, separating functionality at the wrong boundaries can result in high coupling and complexity between features even though the contained functionality within a feature does not significantly overlap.
* **Single Responsibility principle**. Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.
* **Principle of Least Knowledge**(also known as the Law of Demeter or LoD). A component or object should not know about internal details of other components or objects.
* **Don’t repeat yourself (DRY)**. You should only need to specify intent in one place. For example, in terms of application design, specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.
* **Minimize upfront design.**Only design what is necessary. In some cases, you may require upfront comprehensive design and testing if the cost of development or a failure in the design is very high. In other cases, especially for agile development, you can avoid big design upfront (BDUF). If your application requirements are unclear, or if there is a possibility of the design evolving over time, avoid making a large design effort prematurely. This principle is sometimes known as YAGNI ("You ain’t gonna need it").

When designing an application or system, the goal of a software architect is to minimize the complexity by separating the design into different areas of concern. For example, the user interface (UI), business processing, and data access all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not mix code from other areas of concern. For example, UI processing components should not include code that directly accesses a data source, but instead should use either business components or data access components to retrieve data.

However, you must also make a cost/value determination on the investment you make for an application. In some cases, you may need to simplify the structure to allow, for example, UI data binding to a result set. In general, try to consider the functional boundaries from a business viewpoint as well. The following high-level guidelines will help you to consider the wide range of factors that can affect the ease of designing, implementing, deploying, testing, and maintaining your application.

## Design Practices

* **Keep design patterns consistent within each layer**. Within a logical layer, where possible, the design of components should be consistent for a particular operation. For example, if you choose to use the Table Data Gateway pattern to create an object that acts as a gateway to tables or views in a database, you should not include another pattern such as Repository, which uses a different paradigm for accessing data and initializing business entities. However, you may need to use different patterns for tasks in a layer that have a large variation in requirements, such as an application that contains business transaction and reporting functionality.
* **Do not duplicate functionality within an application**. There should be only one component providing a specific functionality—this functionality should not be duplicated in any other component. This makes your components cohesive and makes it easier to optimize the components if a specific feature or functionality changes. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.
* **Prefer composition to inheritance**. Wherever possible, use composition over inheritance when reusing functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes. This also reduces the inheritance hierarchies, which can become very difficult to deal with.
* **Establish a coding style and naming convention for development**. Check to see if the organization has established coding style and naming standards. If not, you should establish common standards. This provides a consistent model that makes it easier for team members to review code they did not write, which leads to better maintainability.
* **Maintain system quality using automated QA techniques during development**. Use unit testing and other automated Quality Analysis techniques, such as dependency analysis and static code analysis, during development. Define clear behavioral and performance metrics for components and sub-systems, and use automated QA tools during the build process to ensure that local design or implementation decisions do not adversely affect the overall system quality.
* **Consider the operation of your application**. Determine what metrics and operational data are required by the IT infrastructure to ensure the efficient deployment and operation of your application. Designing your application’s components and sub-systems with a clear understanding of their individual operational requirements will significantly ease overall deployment and operation. Use automated QA tools during development to ensure that the correct operational data is provided by your application’s components and sub-systems.

## 12 Factor Applications

**All VA Apps Should Follow “12 Factor App” Principles**

The twelve-factor app is a methodology for building web or software-as-a-service apps that:

* Use **declarative** formats for setup automation, to minimize time and cost for new developers joining the project;
* Have a **clean contract** with the underlying operating system, offering **maximum portability** between execution environments;
* Are suitable for **deployment** on modern **cloud platforms**, obviating the need for servers and systems administration;
* **Minimize divergence** between development and production, enabling **continuous deployment** for maximum agility;
* And can **scale up** without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

<http://12factor.net/>

[I. Codebase](http://12factor.net/codebase)

One codebase tracked in revision control, many deploys

[II. Dependencies](http://12factor.net/dependencies)

Explicitly declare and isolate dependencies

[III. Config](http://12factor.net/config)

Store config in the environment

[IV. Backing Services](http://12factor.net/backing-services)

Treat backing services as attached resources

[V. Build, release, run](http://12factor.net/build-release-run)

Strictly separate build and run stages

[VI. Processes](http://12factor.net/processes)

Execute the app as one or more stateless processes

[VII. Port binding](http://12factor.net/port-binding)

Export services via port binding

[VIII. Concurrency](http://12factor.net/concurrency)

Scale out via the process model

[IX. Disposability](http://12factor.net/disposability)

Maximize robustness with fast startup and graceful shutdown

[X. Dev/prod parity](http://12factor.net/dev-prod-parity)

Keep development, staging, and production as similar as possible

[XI. Logs](http://12factor.net/logs)

Treat logs as event streams

[XII. Admin processes](http://12factor.net/admin-processes)

Run admin/management tasks as one-off processes

**Application Layers**

* **Separate the areas of concern**. Break your application into distinct features that overlap in functionality as little as possible. The main benefit of this approach is that a feature or functionality can be optimized independently of other features or functionality. In addition, if one feature fails, it will not cause other features to fail as well, and they can run independently of one another. This approach also helps to make the application easier to understand and design, and facilitates management of complex interdependent systems.
* **Be explicit about how layers communicate with each other**. Allowing every layer in an application to communicate with or have dependencies upon all of the other layers will result in a solution that is more challenging to understand and manage. Make explicit decisions about the dependencies between layers and the data flow between them.
* **Use abstraction to implement loose coupling between layers**. This can be accomplished by defining interface components such as a façade with well known inputs and outputs that translate requests into a format understood by components within the layer. In addition, you can also use Interface types or abstract base classes to define a common interface or shared abstraction (dependency inversion) that must be implemented by interface components.
* **Do not mix different types of components in the same logical layer**. Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers. For example, the UI layer should not contain business processing components, but instead should contain components used to handle user input and process user requests.
* **Keep the data format consistent within a layer or component**. Mixing data formats will make the application more difficult to implement, extend, and maintain. Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation and incur a processing overhead.

**Components, Modules, and Functions**

* **A component or an object should not rely on internal details of other components or objects**. Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if appropriate, how to route it to appropriate subcomponents or other components. This helps to create an application that is more maintainable and adaptable.
* **Do not overload the functionality of a component**. For example, a UI processing component should not contain data access code or attempt to provide additional functionality. Overloaded components often have many functions and properties providing business functionality mixed with crosscutting functionality such as logging and exception handling. The result is a design that is very error prone and difficult to maintain. Applying the single responsibility and separation of concerns principles will help you to avoid this.
* **Understand how components will communicate with each other**. This requires an understanding of the deployment scenarios your application must support. You must determine if all components will run within the same process, or if communication across physical or process boundaries must be supported—perhaps by implementing message-based interfaces.
* **Keep crosscutting code abstracted from the application business logic as far as possible**. Crosscutting code refers to code related to security, communications, or operational management such as logging and instrumentation. Mixing the code that implements these functions with the business logic can lead to a design that is difficult to extend and maintain. Changes to the crosscutting code require touching all of the business logic code that is mixed with the crosscutting code.
* **Define a clear contract for components**. Components, modules, and functions should define a contract or interface specification that describes their usage and behavior clearly. The contract should describe how other components can access the internal functionality of the component, module, or function; and the behavior of that functionality in terms of pre-conditions, post-conditions, side effects, exceptions, performance characteristics, and other factors.

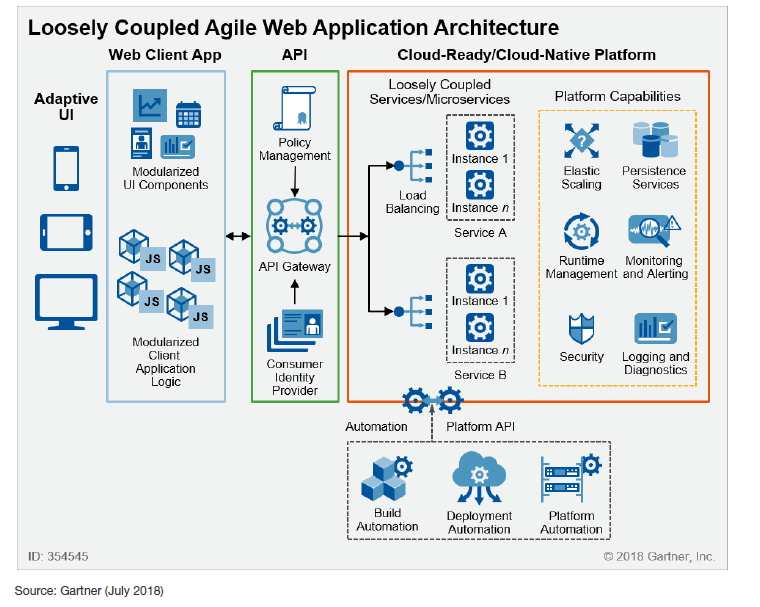
**Product Engineering Guidance**

* Capacity Planning
  + Workload – # of users or transactions/per hour or day
  + Performance – # of seconds per transaction; # of seconds per calculation
  + Security and Privacy – required compliance with VA Directive 6500
* System Architecture
  + Keep track, using eMASS and SNOW, the details of hardware, software, and network architectures are expected. Diagrams and/or descriptions of each are helpful in completing the DIBR and SSP.
  + For hardware architecture, what will the hardware landscape look like? Example: Linux servers will connect to desktops through the firewall, while accessing database servers internally
  + For software architecture, what applications will interact together? What databases (existing and/or new) will be accessed for data retrieval/storage?
  + For network architecture, what will the network connections look like?
* Database Management System
  + What is the Conceptual Data Model? A tree of data elements and their relationship with each other.
  + What database files will be modified and/or created? Identify affected existing database files
  + Another check is to see if project team will be using an authoritative VA data source/model or deriving a data model, based on the Enterprise Logical Data Model (ELDM).
  + Interface Detailed Design
    - External Interface – what are the planned interface(s), what are the details of those, and the system(s) to which they will interact? Have the proper security controls been considered or are they in place, where applicable? If VA sensitive data is to be transferred and/or stored, what are the controls? (address VA Directive 6500, FIPS 140-2, etc.)
    - Human-machine Interface – if graphical user interfaces are to be developed, are they 508 compliant? Are the inputs and outputs known?
  + Security and Privacy
    - The project team should address any logging capabilities; user access, with planned method for authentication and authorization; and use of Identity Access Management (IAM) or equivalent.
    - What are the inherited controls using the VAEC?
    - How will VA-sensitive data, if any, be protected (PHI and PII)?

High Availability and Security Best Practices for Cloud-based Systems:

* Multi-data center architecture intended for high availability
* Isolation of instances between private/public subnets
* Security groups limiting access to only necessary services
* Network access control list (ACL) rules to filter traffic into subnets as an additional layer of network security
* A secured bastion host instance to facilitate restricted login access for system administrator actions
* Standard IAM policies with associated groups and roles, exercising least privilege
* Monitoring and logging; alerts and notifications for critical events
* Use object storage such as S3 (with security features enabled) for logging, archive, and application data
* Implementation of proper load balancing and Auto Scaling capabilities (or use of VM scale sets if using Azure)
* HTTPS-enabled load balancers with hardened security policy
* Leverage approved database backup and encryption

Gartner: Application Architecture Pattern



DevSecOps Best Practices from BIP

The following best practices are followed, as applicable to the Product:

* Incorporate Configuration Management to support the development methodology in all environments
* Build and maintain DevSecOps pipelines for implementing CI/CD for cloud applications
* Automate all phases of software delivery via an instance of a CI/CD pipeline derived from an enterprise pipeline framework. The phases shall support functional and regression testing, including test data conditioning, and incorporate at a minimum:
  + Automated Unit Testing
  + Automated Builds
  + Automated Functional Testing
  + Automated Performance Testing
  + Automated Resiliency Testing
  + Automated Section 508 Compliance Testing
  + Automated Static and Dynamic Security Testing (e.g. Fortify)
  + Automated zero downtime deployments
  + Automated rollback
  + Automated deployment validation
* Develop a CI/CD procedure to support multiple technology stacks including (but not limited to) Java, JavaScript frameworks, Salesforce, SQL, Python, Bash, and Groovy
* Develop a CI/CD procedure to ensure that the application meets the security standards set by the Enterprise
* Develop a CI/CD procedure to following Infrastructure as Code principles and ensure that all physical resources adhere to cloud compliance standards set by the Enterprise
* Develop a CI/CD procedure to support end-to-end business processing and automated testing to support Test Driven Development (TDD) methodologies
* Operate using Monitoring as Code principles and ensure that observability of the application across all layers (infrastructure, OS, application etc.) is “baked into” the provisioning and deployment
* Capture DevSecOps metrics from the CI/CD pipeline for each application to demonstrate compliance along with performance metrics
* Provide administrative support functions for product application and COTS products (e.g. application server, web server, content server, etc.) and middleware platforms like WebLogic and web servers like Apache
* Provide administrative support for Windows and Linux systems
* Identify and institute metrics to measure DevSecOps maturity and automate the collection of the same
* Implement application specific DevSecOps dashboards that enable visualization of the metrics captured by the pipeline
* Professional services provide expertise in DevSecOps tools (Example: Jenkins, Circle CI, GitHub, Nexus, Artifactory, Docker, Kubernetes, Sonar, Maven, Node package manager, Cloud Formation Templates, Terraform, Elastic Stack, App Dynamics)
* Facilitate migration of DevSecOps functions from one tool to another (Ex. From Rational to another Application Lifecycle Management tool)
* Manage government approved cloud services and resources including create and maintain multiple environments in the VAEC, such as Dev (main line), Dev (point releases), Test (main line), Test (point releases), partner integration, performance testing, Software Quality Assurance (SQA), Staging, and Production.
  + Apply OS and application patches.
  + Monitor certificate expiration dates (and renew / update certificates accordingly).
  + Monitor middleware / COTS / open source software releases and schedule required upgrades.
* Define, develop and deliver Monitoring and DevSecOps Dashboards
* Deliver dashboard metrics at the regular agreed upon cadence including the following data:
  + Release cadence and % of automated releases
  + Percent automated test coverage across all testing phases
  + Downtime incurred during deployments
  + Release Failure Rate
  + Lead time
  + Meantime to restore
  + Successful automated rollbacks
  + Percentage of time system meets or exceeds performance, capacity, and availability targets
  + Other metrics as requested by the VA PM
* Define, develop and deliver application monitoring plan, including dashboards
* Define, develop and deliver Automated Testing Reports
* Perform activities associated with responding to incidents, repairing defects, and analyzing, designing, developing, implementing and maintaining minor functional and/or technical enhancements, and/or initiating and applying refactoring to applications in production to improve the performance and/or stability of an application.
* Perform activities required for providing, maintaining, securing, scheduling, backing up, recovering and supporting the Product environments.

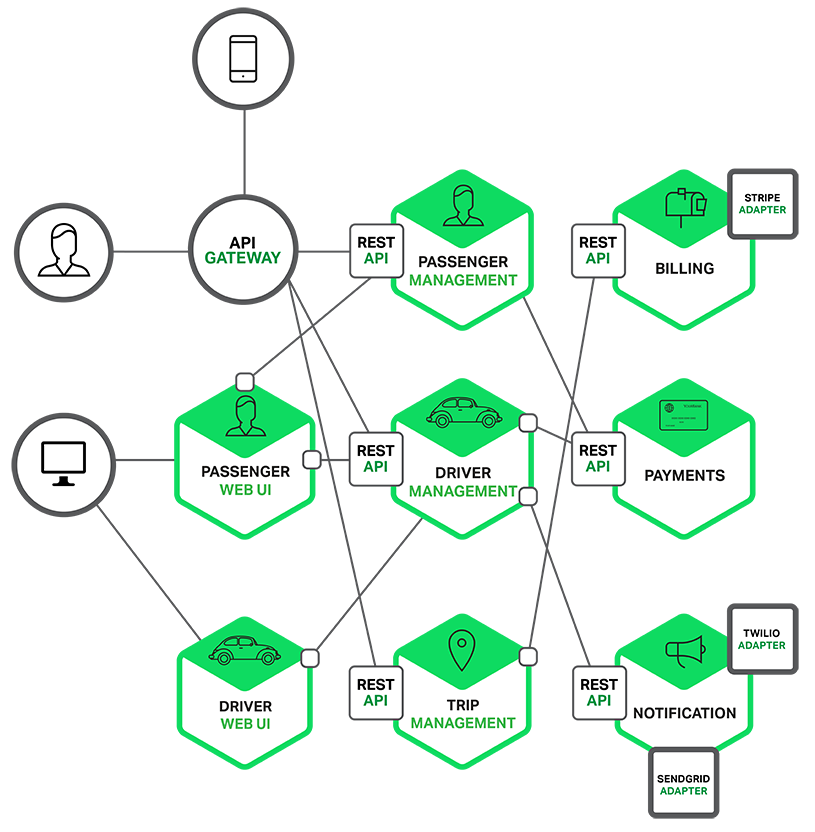
Moving to Microservices

Source: DoD Enterprise DevSecOps Initiative Moving to Microservices Document v1.3 (2020)

A 'microservice' is a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services.

In a Microservices architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop, test, and more resilient to architecture erosion. It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.

It also allows the architecture of an individual service to emerge through continuous refactoring. Microservices-based architectures enable continuous delivery and deployment.



## Strangler Pattern

Martin Fowler describes the Strangler Application:

* One of the natural wonders of this area are the huge strangler vines. They seed in the upper branches of a fig tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host.
* To get there, the following steps were followed:
  + First, add a proxy, which sits between the legacy application and the user. Initially, this proxy doesn’t do anything but pass all traffic, unmodified, to the application.
  + Then, add new service (with its own database(s) and other supporting infrastructure) and link it to the proxy. Implement the first new page in this service. Then allow the proxy to serve traffic to that page (see below)
  + Add more pages, more functionality and potentially more services. Open up the proxy to the new pages and services. Repeat until all required functionality is handled by the new stack.
  + The monolith no longer serves traffic and can be switched off.

Learn more: <https://www.ibm.com/developerworks/cloud/library/cl-strangler-application-pattern-microservices-apps-trs/index.html> and <https://www.michielrook.nl/2016/11/strangler-pattern-practice/>

## Key Decisions

* Infrastructure as Code: all configs, including RBAC, SDN, load balancing etc., should be in code
* Programming language per microservices (couple of options such as Java, Python, Go…). Select one framework per programming language like Spring for example.
* Databases per service
* Encrypted by default (part of Service Mesh usually for west/east traffic)
* Transport (RPC/HTTPs etc.)
* API Gateway (understand difference between internal/external traffic mesh vs gateway) (can be provided by Service Mesh or Kubernetes)
* Service Mesh
* Authentication/Authorization (JSON Web Token (JWT) etc.) (part of Service Mesh or API Gateway usually). Please note that you will also create your own auth microservice for your internal fine grained auth.
* Access Control (ACL) (part of Service Mesh usually)
* Messaging
* Storage management (for persistent storage)
* Health/Readiness (part of Kubernetes/Docker)
* Logs (part of Kubernetes)
* Telemetry (understand the difference with traditional logs)
* Monitoring solution
* Observability (Tracing)
* Circuit breaker (part of Service Mesh usually)
* Use bounded retries and timeouts (part of Service Mesh usually)
* High Availability (part of Kubernetes and/or Service Mesh)
* Load Balancing (part of Kubernetes and/or Service Mesh)
* Service Discovery (part of Kubernetes and/or Service Mesh)
* Canary / Traffic Management (part of Service Mesh usually)
* Backups (storage + containers + kubernetes config)
* Key Management and Certificate management (part of Service Mesh usually)
* Centralized secrets management (usually managed by Kubernetes)

Note: Kubernetes with ISTIO as a Service Mesh can provide a lot of these features by DEFAULT. These are included with IBM Cloud Paks as part of VA Platform One (VAPO).

See: <https://istio.io/docs/concepts/security/> and more at istio.io

## Recommendations

* First step: define bounded context! Understand bounded context: <https://martinfowler.com/bliki/BoundedContext.html>
* Stateless
* Cattle not pets!
* **Separate data store (database) per microservice**
* Self-contained services
* Loosely coupled
* Ensure your services are idempotent
* Define which service can be asynchronous vs synchronous.
* Understand API Gateway vs Service Mesh and clearly define Authentication/Authorization process between services and User auth.
* Understand JSON Web Tokens to achieve the end goal of creating a distributed authentication mechanism for Microservices <https://nordicapis.com/how-to-control-user-identity-within-microservices/>.
* Understand volume of communications between two services to verify coupling.
* Leverage DDD (Domain driven design) <https://en.wikipedia.org/wiki/Domain-driven_design>
* 2 pizza team (separate team for each microservice)
* Use REST whenever possible
* Use JWT at the Gateway/Mesh layer for centralized auth.
* Separate Shared Libraries from Microservices
* Leverage messaging and asynchronous communication when possible
* Use event-driven architecture to ensure that when something happens to one service, the other services can follow up with their own actions.
* Check your inter service communication volumes, if too high, you have a coupling issue.
* Recommend sticking to one or two programming languages as it can create complexity within teams
* Leverage Zero Trust model (all blocked by default) and whitelisting. (part of Service Mesh usually)
* Deploy Microservices in containers
* Keep in mind graceful failure is key and avoid single points of failure!
* Recommend using micro segmentation to segment Microservices clusters. (part of Service Mesh usually)
* Use whitelisting for access and NOT blacklisting using fine grained role based access control (RBAC) (part of Service Mesh usually)
* Have a CI/CD pipeline per microservice with separate builds per microservice. You can use the same DevSecOps platform for all services. Eg. Same Jenkins but a Jenkinsfile per microservice.
* If moving from Legacy to Microservices, use the Strangler Pattern – Learn more: <https://www.ibm.com/developerworks/cloud/library/cl-strangler-application-pattern-microservices-apps-trs/index.html> and <https://www.michielrook.nl/2016/11/strangler-pattern-practice/>
* Leverage DevSecOps and CI/CD with DoD hardened containers
* Leverage Chaos concepts to try to break things before they actually really break!
* Use JSON instead of XML whenever possible
* Use YAML for configurations
* Use caching when possible.
* Each service should do its authorization but global authentication can be used.
* Use Alerts solutions such as Prometheus to automate health/issue detection
* Leverage Tracing to track requests through multiple services and properly debug issues

## Read more

* <https://microservices.io/patterns/microservices.html>- microservices patterns
* <https://dzone.com/articles/top-5-microservices-architecture-and-design-best-p>- microservices architecture
* <https://medium.freecodecamp.org/follow-these-practical-principles-and-get-well-designed-microservices-boundaries-ef2deffd69e3> - how to define Microservices boundaries
* <https://medium.com/microservices-in-practice/microservices-in-practice-7a3e85b6624c> - microservices in practice
* <https://nordicapis.com/how-to-control-user-identity-within-microservices/> - how to control user identity
* https://medium.com/technology-learning/how-we-solved-authentication-and-authorization-in-our-microservice-architecture-994539d1b6e6 - how solve auth
* <https://auth0.com/blog/introduction-to-microservices-part-4-dependencies/> - how to share data between services
* <http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/> Microservices architecture