# Key Design Principles

When getting started, keep in mind the key principles that will help create a technical 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 general, try to consider the functional boundaries from a business viewpoint as well. These guidelines will help consider the range of factors that can affect the ease of designing, implementing, deploying, testing, and maintaining your application.

For more information, please refer to the Microsoft Application Architecture Guide. The design principles and patterns described in this guide are also applicable to other technologies. <https://docs.microsoft.com/en-us/azure/architecture/guide/>

# 12 Factor Design Patterns

<http://12factor.net/> The twelve-factor app is a methodology for building 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 (design patterns) can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc.).

[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

# High Availability and Security Design Patterns

* 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

Note: These patterns originated from AWS cloud systems but apply to systems on other platforms including on-premises hosting environments.

More information:

[Enterprise Technology Guidelines - Dept of Veterans Affairs-External - Confluence (va.gov)](https://confluence.devops.va.gov/pages/viewpage.action?pageId=95763599)

# DevSecOps Design Patterns from BIP

* 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.)
* 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, 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 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 and repairing defects.
* Perform activities required for providing, maintaining, securing, scheduling, backing up, recovering, and supporting the Product environments.

Additional references on Design Patterns:

[DevSecOps Best Practices Repository - Home (sharepoint.com)](https://dvagov.sharepoint.com/sites/OITEPMOVAEA/DevSecOps/SitePages/Home.aspx?RootFolder=%2Fsites%2FOITEPMOVAEA%2FDevSecOps%2FShared%20Documents%2FDevelopment%20and%20Testing%20Best%20Practices&FolderCTID=0x0120004E8B4A0F76D0C644B44E3A90A3ECBAEC&View=%7B2699C717%2DE900%2D4DD9%2D9B32%2DECFA788183AC%7D)

# Microservices Architecture Patterns

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 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.

**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)

**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. E.g., 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>
  + <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 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

# 14 Software Architecture Patterns (Red Hat)

[14 software architecture design patterns to know | Enable Architect (redhat.com)](https://www.redhat.com/architect/14-software-architecture-patterns)

The [**circuit breaker**](https://www.redhat.com/architect/circuit-breaker-architecture-pattern) pattern minimizes the effects of a hazard by rerouting traffic to another service. While it helps make systems more fault tolerant to prevent accidents, it also requires sophisticated testing and using an infrastructure-management technology like service mesh.

The [**client-server**](https://www.redhat.com/architect/5-essential-patterns-software-architecture#client-server) pattern is a peer-to-peer architecture that is comprised of a client, which requests a service, and a server, which provides the service. Examples include banking, file sharing, email, and the World Wide Web. One advantage of this pattern is that data and network peripherals are centrally managed, however, the server is expensive.

The [**command query responsibility segregation**](https://www.redhat.com/architect/pros-and-cons-cqrs) (CQRS) pattern handles the situation where database queries happen more often than the data changes. It separates read and write activities to provide greater stability, scalability, and performance, but it requires more database technologies and therefore may increase costs.

The [**controller-responder**](https://www.redhat.com/architect/5-essential-patterns-software-architecture#controller-responder) pattern divides the architecture into two components: The controller handles the data and distributes workloads, and the responder replicates data from the controller and generates results. One advantage is that you can read data from the responder without affecting the data in the controller, but if the controller fails, you may lose data and need to restart the application.

The [**event sourcing**](https://www.redhat.com/architect/pros-and-cons-event-sourcing-architecture-pattern) pattern is good for applications that use real-time data. It sends a continuous stream of messages to a database, web server, log, or another target. It's very flexible but demands a highly efficient and reliable network infrastructure to minimize latency.

The [**layered**](https://www.redhat.com/architect/5-essential-patterns-software-architecture#layered) pattern is good for e-commerce, desktop, and other applications that include groups of subtasks that execute in a specific order. The layered pattern makes it easy to write applications quickly, but a disadvantage is that it can be hard to split up the layers later.

The [**microservices**](https://www.redhat.com/architect/5-essential-patterns-software-architecture#microservices) pattern combines design patterns to create multiple services that work interdependently to create a larger application. Because each application is small, it's easier to update them when needed, but the complexity means you need greater architectural expertise to make everything work correctly.

The [**model-view-controller**](https://www.redhat.com/architect/5-essential-patterns-software-architecture#MVC) (MVC) pattern divides an application into three components. The model contains the application's data and main functionality; the view displays data and interacts with the user; and the controller handles user input and acts as the mediator between the model and the view. This pattern enables the application to generate various views, but its layers of abstraction increase complexity.

**[ Learn more about**[***validated patterns***](https://www.redhat.com/en/topics/cloud-computing/what-are-validated-patterns?intcmp=7013a0000025wJwAAI)**. ]**

The [**pub-sub**](https://www.redhat.com/architect/pub-sub-pros-and-cons) pattern sends (publishes) relevant messages to places that have subscribed to a topic. It's easy to configure but more challenging to test because interactions between the publisher and the subscriber are asynchoronous.

The [**saga**](https://www.redhat.com/architect/pros-and-cons-saga-architecture-pattern) pattern is used for transactions with multiple steps, such as travel reservation services. A "saga" includes the various steps that must happen for the transaction to complete. This pattern enables transactions (ideally with five or fewer steps) to happen in loosely coupled, message-driven environments, but it requires a lot of programming and can be complex to manage.

The [**sharding**](https://www.redhat.com/architect/pros-and-cons-sharding) pattern segments data in a database to speed commands or queries. It ensures storage is consumed equally across instances but demands a skilled and experienced database administrator to manage sharding effectively.

The [**static content hosting**](https://www.redhat.com/architect/pros-and-cons-static-content-hosting-architecture-pattern) pattern is used to optimize webpage loading time. It stores static content (information that doesn't change often, like an author's bio or an MP3 file) separately from dynamic content (like stock prices). It's very efficient for delivering content and media that doesn't change often, but downsides include data consistency and higher storage costs.

The [**strangler**](https://www.redhat.com/architect/pros-and-cons-strangler-architecture-pattern) pattern is used when you're making incremental changes to a system. It places the old system behind an intermediary to support incremental transformation, which reduces risk compared to making larger changes. However, you need to pay close attention to routing and network management and make sure you have a rollback plan in place in case things go wrong.

The [**throttling**](https://www.redhat.com/architect/pros-and-cons-throttling) (or rate-limiting) pattern controls how fast data flows into a target. It's often used to prevent failure during a distributed denial of service attack or to manage cloud infrastructure costs. To use this pattern successfully, you need good redundancy mechanisms in place, and it's often used alongside the circuit breaker pattern to maintain service performance.