# Microservices Architecture Pattern

A Proposed Architectural Pattern to Engineer Trustworthy Secure Systems

CSA cloud security alliance®

The permanent and official location for Applications Containers and Microservices Working Group is https://cloudsecurityalliance.org/research/working-groups/containerization/

# Table of Contents

# Acknowledgments

## Editors/ Working Group Co-Chairs:

Anil Karmel
Andrew Wild

## Primary Contributors:

Gustavo Nieves Arreaza
Marina Bregkou
Craig Ellrod
Michael Holden
John Jiang
Kevin Keane
Vani Murthy
Pradeep Nambiar
Vinod Babu Vanjarapu
Mark Yanalitis

## Reviewers:

Ankur Gargi
Alex Rebo
Michael Roza
Ankit Sharma

## CSA Analysts:

Hillary Baron
Marina Bregkou

The mission of the CSA Application Containers and Microservices (ACM) working group is to conduct research on the security of application containers and microservices and publish guidance and best practices for the secure use of application containers and microservices. This working group is building upon the work done by the joint NIST/CSA Applications Container Security Guidance developed by the NIST Cloud Security Working Group.

# 1. Introduction

The impact of weakly constructed microservices exists[1], realized as under-secured and over-exposed application programming interfaces (API's) which form the core tranche of microservice application risk. Some business and technology owners skip the architecture methods[2] and search for a software solution primarily based upon a set of coarse-grained requirements. Those that research solutions in the open market end up applying architecture methods to fit the purchased solution into the existing control landscape. Even newly constructed greenfield microservice applications integrate with the rest of the legacy enterprise - companies do not throw out the existing architecture each year. Inevitably, product purchases without architecture methods lead to trade-offs capable of introducing constraints and add-ons at a later date when the monetary cost to modify increases over time.

Whether a business leader has a bias to purchasing solutions or supports a "build in-house" mindset, API consumption and microservice integration is a common systems integration expectation. It is best to have a means to guide integration with the use of architecture, architecture patterns, and security control overlays to ensure that information security is an upfront requirement set. Microservice architectural patterns and corresponding security control overlays set the foundation for microservices development as a complete thought. Patterns plus overlays ensure baked-in information security. Done correctly, security control overlays become indistinguishable from the architecture and design methods used to create microservice applications. Some would call this phenomena DevSecOps[3]. The security control overlay concept originated in the Federal Information Systems Management Act (FISMA) Project[4]. As per the FISMA project, *"An overlay is a fully-specified set of controls, control enhancements, and supplemental guidance derived from the application of tailoring guidance to control baselines."* Security controls overlays are further elaborated in detail in National Institutes of Standards Special Publication Security and Privacy Controls for Federal Information Systems and Organizations SP800-53 version 4 Section 3.3[5]. Although overlay is a NIST-introduced concept, other control frameworks, like ISACA COBIT 5, PCI-DSS 3.2.1, or CSA CCM v3.0.1, can be used.

[1] Hinkley, C. (2019, November 6). *Dissecting the Risks and Benefits of Microservice Architecture.* TechZone360. https://www.techzone360.com/topics/techzone/articles/2019/11/06/443660-dissecting-risks-benefits-microservice-architecture.htm

[2] The Open Group. *The TOGAF Standard, Version 9.2 Overview, Phase A, B, C, D, E, F, and G.* Retrieved August 11, 2021, from https://www.opengroup.org/togaf.

[3] Cloud Security Alliance. (2019, August 1). *Information Security Management through Reflexive Security.* https://cloudsecurityalliance.org/artifacts/information-security-management-through-reflexive-security/ (13, 14, 16)

[4] NIST. *Security and Private Control Overlay Overview.* Retrieved August 11, 2021, from https://csrc.nist.gov/projects/risk-management/sp800-53-controls/overlay-repository/overlay-overview

[5] NIST. (2020, September). *NIST Special Publication 800-53, Revision 5: Security and Privacy Controls for Information Systems and Organizations*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf

Software design patterns guide software development[6]. Security control overlays, a discrete collection that represents a fully-specified set of controls, control enhancements, and supplemental guidance, integrates into the architectural design process as embedded and upfront administrative, technical or physical requirements. Together, software design patterns and security control overlays inform software development to "build in" security as a design element, and not as an application of facade applied at the end where changes to software are most expensive to make. The following pages establish the foundation for software architecture as a complete thought. *A complete thought in an architectural sense* is an architectural expression that behaves like a good math equation; it can be run forward to its expected product and backward to its non-functional and functional requirements.

Once practitioners master software design patterns and security control overlays, use unlocks additional levels of architectural maturity where specific microservice viewpoints reveal needed control state in specific aspects of the underlying service delivery framework such as the data, integration, and deployment architectures.  Although not "microservice architectures," they are supporting architectures and designs requiring assurance that system behavior remains repeatable, reliable, accurate, and complete according to an understood control state.

Microservice architecture style represents a distributed application processing footprint where the combined capability derived from static configuration, dynamic adaptation, and abstraction yield what people would term "application functionality." Prior to the microservices and containerization transformation, many configurations and abstractions existed within a singular monolithic application boundary. As monolithic code bases became larger, internal application states and co-dependencies became increasingly difficult to discern leading to a number of architecture, development, and operations challenges. However, it remained common to have one business representative  accountable for business process functionality, one product owner fulfilling application custodial obligations, and managing allied developers housed under a singular organization structure. Microservices architecture style changes organization structure just as much as it changes software construction and assembly. Each part of the architecture, whether at the platform, software-defined, application programming interface (API), or software development level, performs a specific and particular function with respect to the entire functionality delivered by a microservice application. Organizations responded to virtualization of compute, memory, storage, and network into a single capability, by combining teams to match the technological change. Hybrids of storage, network, and server computer teams materialized from the combination of discrete teams.

As microservice software development takes root, the industry is placing an increasing emphasis on DevSecOps[7], and federation of software assembly as well as application security. For example, a business owner may own the entirety of a workflow application, but only deal with the forward-facing requests for improvements to existing capability or new capability. *A business owner concerns themselves with maximizing the value of delivered or deliverable results;* this role exists outside the teams building the software. In the microservice architectural style, it is possible that multiple

[6] Bass, L., Clements, P. C., & Kazman, R. (2012, September). *Software Architecture in Practice, Third Edition.* https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264

[7] Cloud Security Alliance. (2019, August 7). *Six Pillars of DevSecOps.* https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/. 5-6.

product owners align with a singular business owner. *A product owner and associated software development team (or "crew" in Agile parlance) may have ownership of specific functionality,* such as search functionality (API usage, sort, algorithms, metadata cataloging), while a second product owner focuses on the front-end customer facing user experience (Style, presentation, flow and overall experience). Data integration may be left to an allied crew that serves multiple product owner needs for data. Monolithic applications bound all of these roles and actions into a vertical of support, which is a key microservice architectural style distinction - the microservice software development and production deployment behavior flattens an organization's traditional verticals. Not only is the application functionality distributed, so is its support structure forcing a greater reliance on automation to deliver a wide variety of infrastructure, policy, security, identity, and network capability previously isolated into verticals. *Operations typically own the set of processes to deploy and manage IT services,* however the deployment and testing responsibility at times shifts-left *toward the developer who builds the software.* Out of necessity, *architects commonly take on new skill sets in software engineering, to better translate customary and traditional architectural worksets into the microservice architectural style.*

Appendix B further defines the roles of business owner, product owner, developer, operator and architect. For further detail and specific definition of these five roles, please refer to the glossary.

# 2. Goal

In the CSA, February 2020 publication, "Best Practices in Implementing a Secure Microservices Architecture"[8], readers were presented with guidance about the engineering of trustworthy secure systems, with the last chapter focused through the lens of the Developer, the Operator and the Architect.

This document serves to propose a repeatable approach to architecting, developing and deploying Microservices as a "MAP" (Microservices Architecture Pattern). The proposed MAP contains all the information necessary for a microservice to operate independently and communicate with other microservices which, in aggregate, become capabilities which, in turn, become the components of an application. This paper describes the key elements of the MAP, how they should be designed and deployed, shifting security & compliance left via a continuous compliance-as-code approach.

The primary goal of this work effort is to develop a vendor neutral reference architecture foundation that decomposes into software architecture patterns represented in Software and Platform (Enterprise) Planes, and then can be built back up with the addition of security control overlays. This can be demonstrated by the successful decomposition and recomposition of microservice architecture patterns where the integral action is the overlay of security controls.

[8] Cloud Security Alliance. (2020, February 24). Best Practices in Implementing a Secure Microservices Architecture. https://cloudsecurityalliance.org/artifacts/best-practices-in-implementing-a-secure-microservices-architecture/

## 3. Audience

The intended audience for this document is application developers, application architects, system and security administrators, security program managers, information system security officers, and others who have responsibilities for or are otherwise interested in the security of application containers and microservices.

This document assumes that readers have some operating system, networking, and security expertise, as well as expertise with application containers, microservices and agile application development. Because of the constantly changing nature of application container technologies, readers are encouraged to take advantage of other resources, including those listed in this document, for more current and detailed information.

## 4. Architecture vs. Solutions

Architecture is not a solution. A solution is the application of architecture, patterns, and design effort to solve a specific industry need or business problem. A solution intends to provide ongoing customer and business owner value.  For example, Point-of-Sale (POS) systems represent a composite of people interactions, technology enabled business processes, and a payment platform backend designed to create a specific business capability that cannot be satisfied by architecture alone.  The POS solution design is a combination of concepts, system properties, and patterns assembled to solve a particular business challenge. Problems have solutions which solve the crux of a challenge. To understand any problem end-to-end, one must reach back to the conditions and decisions that created the problem.

Therein lies the difference: a solutions' basis is in its design. Designs include an assemblage of patterns, which originates from the earliest form of abstraction, an architecture.  An architecture forms fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.  Business and technical sketches and drawings remain the primary means by which architects communicate their ideas to an engineering design team. To further remove variability from the development process, architects work with engineers to select agreed upon patterns to form the basis of designs, and frame design activity. Architectures, patterns and designs do not reference a specific named technology, remaining vendor neutral. When the design is finalized, business owners and technology owners choose to build all, buy all, or combine both construction approaches to create a business-aligned technology solution to the problem. Some business and technology owners choose to skip the architectural process and search for a solution based upon a set of requirements, others choose to build up to a solution from requirements.

Those that go to market seeking turn-key, vertically integrated, low-code, or no-code solutions end up applying architecture methods to fit the purchased solution into the existing control landscape. Inevitably this leads to trade-offs, some very sensitive to initial conditions, capable of introducing constraints or solution refactoring at a later date when the economic cost to modify increases in monetary value; accumulating technical debt.
Many reference microservices as an architecture, but in reality it is a style of architecture. Concrete

slab Brutalism and Oak-laden Mission Craftsman represent architectural styles capable of rendering a building. Each style possesses construction principles that when applied correctly manifest in blueprints, whose patterns lead to a specific design to follow, which physically renders the building in the expected style. If the goal is to build a highly modular distributed application, applying microservice  principles, architecture, patterns and design lead to a microservice application.

## 4.1 Patterns and Control Overlays

A pattern is a repeatable set of actions and arrangements that occur in a predictable manner. Patterns can be observed based upon physical appearance, direct or indirect observation, or by analysis. When designing application systems, a software pattern is a known reusable approach to a class of computer programming problems. Software patterns show the relationship between construction elements without specifying the final resolution to the problem or requirement. Software patterns can be thought of as an intermediate structural representation between the software code and the systems that support the software. Historically software patterns lack a key macro-architectural representation: security control guidance. Developers generally do not develop their own security solutions opting instead to rely upon inheriting capability from the platform or application, or if compelled - create their own security controls. Using a salted hash function to one-way encrypt stored passwords is one example of a developer-built security control. Historically, software patterns guide the development of software, not the application of information technology (IT) security controls.

An IT security control offers a means to regulate and govern the behavior of an application system. An IT security control is a noun-abstraction of an underlying technical, administrative, or physical capability corresponding to a preventive, detective, or corrective counter-measure to a perceived risk. The goal of an IT security control is to reduce risk potential to an acceptable, innocuous, or inconsequential level. A control objective is also a noun-abstraction but it differs in that it is a statement of the desired result or purpose to be achieved by implementing controls in a particular manner. Achieving a control objective, occurs with the application of a single control, or commonly a collection of controls; this is referred to as defense in depth – the application of multiple controls of different types acting in concert to prevent, detect, and/or correct sub-optimal operational states. Control objectives originate from a control framework, a set of fundamental controls that facilitates the discharge of business process owner responsibilities to prevent information loss. Layers of controls act at different levels of an application's life cycle, as well as at different levels of its environment. IT security controls exist at the genesis of new software, its construction, deployment, and during platform run-time; recognized as a software development life cycle.

Security professionals do push left with the developer but the level of specialization required in the application security space is increasingly different and distant to IT security platform specialists. Organizations manage the skills gap by training or hiring vanguard software developers, or use a combination of roles involving different specialists. It is possible to manage the skills gap under monolithic applications, however the materialization of distributed application design stretches siloed departments and span of control further aggravating skills gaps and ownership.

What is different and least explained or explored, is the space between the enterprise platform plane where multiple organizational silos apply IT controls (Network, Security, Server, Storage, Messaging), and the software development plane (Secure SDLC and security testing automation). Security control overlays link both the enterprise platform plane and the lower software plane together. The use of overlays represents a domain suited to an expanded security architecture role where confidentiality, integrity, and availability expand to include resiliency and privacy concerns. As the world moves to an abstracted software-centric approach and enterprises embrace software-centrism, dependence demands not only service resilience but complete privacy with the ability to self-manage data access over the entire human-computer interaction. Security architecture as a complete thought is person, platform, plus software.

Security Architecture represents the portion of the enterprise architecture that specifically addresses information system resilience and provides architectural information for the implementation of capabilities to meet security requirements. Security architecture in the micro-service context not only introduces the concept of control overlay upon an existing platform and software architecture, the security architect has accountability and responsibility for establishing the distinction between controls overlays manifesting at the platform level, from controls overlays that manifest in the software itself. As a complete security architecture thought, overlays are collections of controls acting as one to reduce risk; they can be a mix of administrative, technical, and physical controls.

A well-rounded security architect will from time to time construct threat models before applying controls to complicated solution architectures. Threat models are a form of mastery. Methods like Red-Blue teaming, STRIDE, attack trees, Trike, VAST, PASTA, and ISO-31010 Delphi are examples of risk identification approaches. Threat models are very much a people-oriented activity. The best analysis materializes from a diverse body of focused people who have deep knowledge of different aspects of the attack surface, far exceeding the singular knowledge of one person. Group analyses are less prone to brittleness in the face of systemic shocks to the analysis. A robust threat model comes from deep industry vertical knowledge of current state, institutional history, imagination, and selecting an analysis method suiting the problem and not the security architect's comfort level. It is important to keep threat models bound to the tactical scope and avoid ethereal, existential, and Black Swan scenarios. Threat model outcomes legitimize the placement of IT security controls, leading to a robust overlay mitigating known or supposed risks. In that respect, security architecture is distinct from solution architecture, as it mitigates or extinguishes technical, administrative, or physical risk potentials found in a proposed solution architecture.

A microservice application will not contain every design pattern and every security controls overlay, but it will contain those software architecture patterns and overlays considered essential to a working design, solving the customer's problem. However, this is where the software plane flattens and merges into the platform plane. In the microservices architecture style, no solution is a complete thought conceptually until each consumed pattern has a corresponding security control overlay. All overlays act in concert with patterns to complete the solution architecture. Together, patterns and overlays comprise the microservice application's security control state.

## 4.2 Introduction to the Microservices Architecture Pattern (MAP)

To guide the application of security controls overlays to microservices, the generic microservice architecture pattern forks into two different viewpoints. The first viewpoint serves as the enterprise plane. The enterprise plane contains Information technology assets assisting and enabling governance of the microservice architecture. The enterprise aspiration is to reduce variability in the control state. Custom coding, production state workarounds, and one-off modifications increase development costs. Technical debt, for example in the form of bolt-on security appliance additions, introduction of tight coupling constraining flexibility in the design, can hamper the ability to deploy infrastructure as code thereby creating persistence when it is not desired. The enterprise context expects software development to inherit as many security controls as possible, to prevent the variability and unreliability materializing when development teams are left to devise their own application security guidance. Security overlays exist both at the enterprise plane as well as the software plane. API registries handle the inventory and versioning of developed microservices and host service providers and 3rd party integrations. Service repositories (API specifications, templates, code scaffolding, artifacts for the build process) create uniformity in the development of microservices, automate the static and dynamic security testing, and also bootstrap microservices into their containers, ultimately delivering combined capability via a common pipeline (for example a Jenkins job that triggers security checks, then a configuration management resource deployment plan). In an ideal state, the Enterprise plane wants to deliver platform level security hooks, calls, and integrations during the bootstrap process to avoid having to make modifications during runtime.

The second viewpoint serves as the software plane. Figure 1 represents the decomposition of a distributed microservice application, which exists in the software plane and is a representation closest to the software code. This figure renders a generic viewpoint of a synthetic distributed microservice application. It serves as an x-ray showing the major inheritable enterprise plane system integrations above and microservice components below. Machine client types (Browser, IoT, Mobile, API integrations) and physical consumers access the features that a distributed microservice application offers via the enterprise plane. The API controls the presentation logic consumed by a client and offers one and only one business function, and contain other business rules that control what the client can elicit from the exposed API. The API can marshall data from multiple sources, and access data on a data volume mounted external to the container hosting the microservice API. Microservices take advantage of enterprise plane services and software plane API gateway services to load balance across containers, permitting more than one instance of a microservice running in production simeotaneously. Within the software plane in the microservice context, sidecar services represent utility services handling tasks unrelated to the business logic found in the microservice. Each microservice API has a sidecar paired with it and in a clustered container environment the sidecar is the means to deliver service to service communications policy and security context policy. The primary traffic shaping mechanism is network access control logic.  Control can come in the form of virtual LAN's, policy based routing, context-based ACL's, specific gateway logic, and software defined networking. Each of these enterprise plane implementations come with security overlay trade-offs. Within the distributed microservice, the primary means to shape traffic and control flow is enterprise plane managed authorization and access tokens carrying the encrypted client entitlements (i.e. oauth), externally managed escrowed credentials (e.g. platform plane machine to machine credentials escrow and management), and mTLS (enterprise plane mutual transport layer security certificates management). Security control overlays act within a plane, and across enterprise and software planes - leading to defense in depth. Security control overlays exist in both planes.

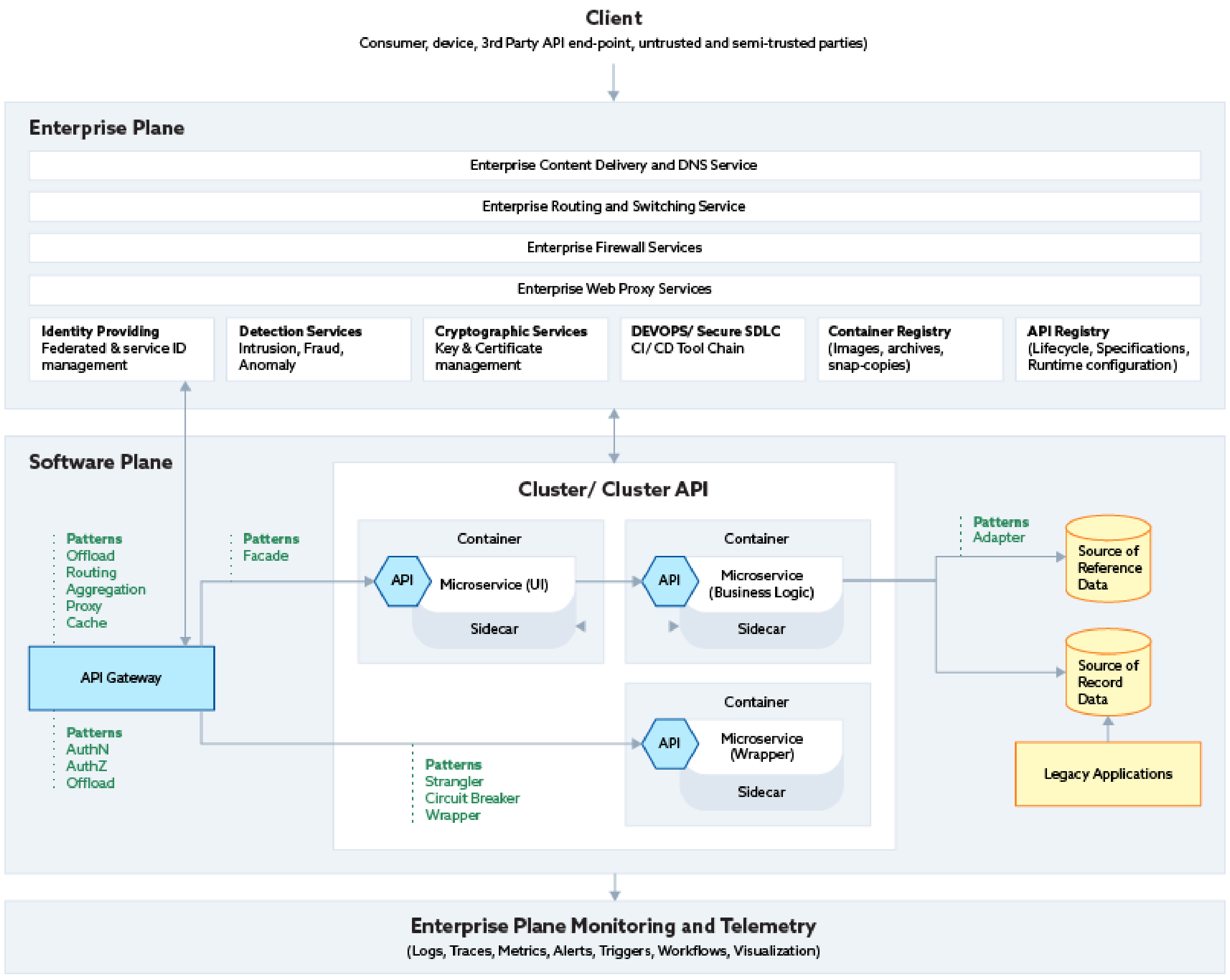

Client
Consumer, device, 3rd Party API end-point, untrusted and semi-trusted parties)
Enterprise Plane
Enterprise Content Delivery and DNS Service
Enterprise Routing and Switching Service
Enterprise Firewall Services
Enterprise Web Proxy Services
Identity Providing Federated & service ID management
Detection Services Intrusion, Fraud, Anomaly
Cryptographic Services Key & Certificate management
DEVOPS/ Secure SDLC CI/ CD Tool Chain
Container Registry (Images, archives, snap-copies)
API Registry (Lifecycle, Specifications, Runtime configuration)
Software Plane
Patterns Offload Routing Aggregation Proxy Cache
Patterns Facade
API Gateway
Patterns AuthN AuthZ Offload
Cluster/ Cluster API
Container
API
Microservice (UI)
Sidecar
Container
API
Microservice (Business Logic)
Sidecar
Container
API
Microservice (Wrapper)
Sidecar
Patterns Strangler Circuit Breaker Wrapper
Patterns Adapter
Source of Reference Data
Source of Record Data
Legacy Applications
Enterprise Plane Monitoring and Telemetry
(Logs, Traces, Metrics, Alerts, Triggers, Workflows, Visualization)

*Figure 1.*
*Microservices Reference Architecture - Enterprise and Software Planes*

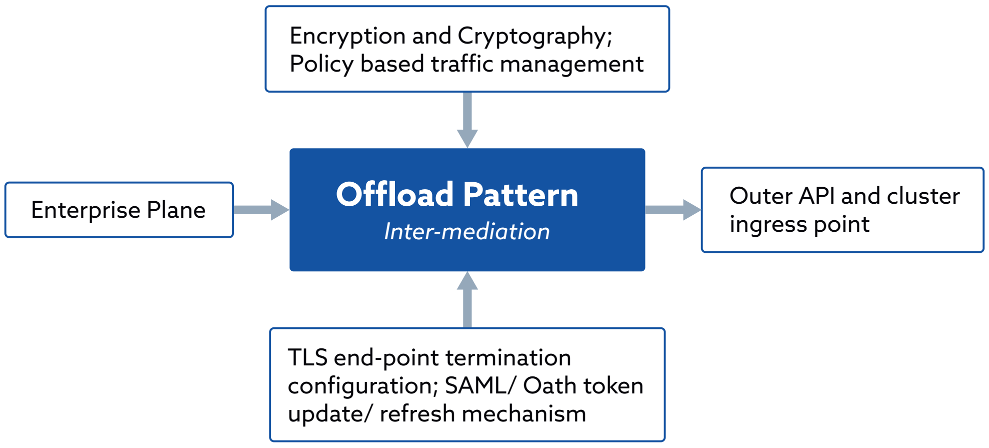
# 5. Microservice Architecture Patterns

## 5.1 Patterns

The following architecture patterns apply to the secure development of microservices into business applications. As you review these patterns, note that many work together to form a secure system. While you may begin with a bias towards a single pattern (e.g., Authentication), it is worthwhile to understand how the patterns interact with each other to support a secure and resilient business solution.

### 5.1.1 Offload Pattern

Offloading is a generic data stream action that is context specific. Offloading can be tightly coupled to API gateway functionality, such as providing TLS cryptographic termination functionality using kernel software or accelerator hardware so back end devices do not have to manage TLS connections.Offloading can also be tightly coupled to the data access layer where data writes are spread across several simultaneous data commit actions to speed up data writing or reading performance.  Offloading can also be applied to authentication and authorization. As a general rule an offloaded function is one that is frequently consumed by many other services as a shared service. If one chooses to offload a service, it represents a resource that a microservice API does not have to include in its' codebase.

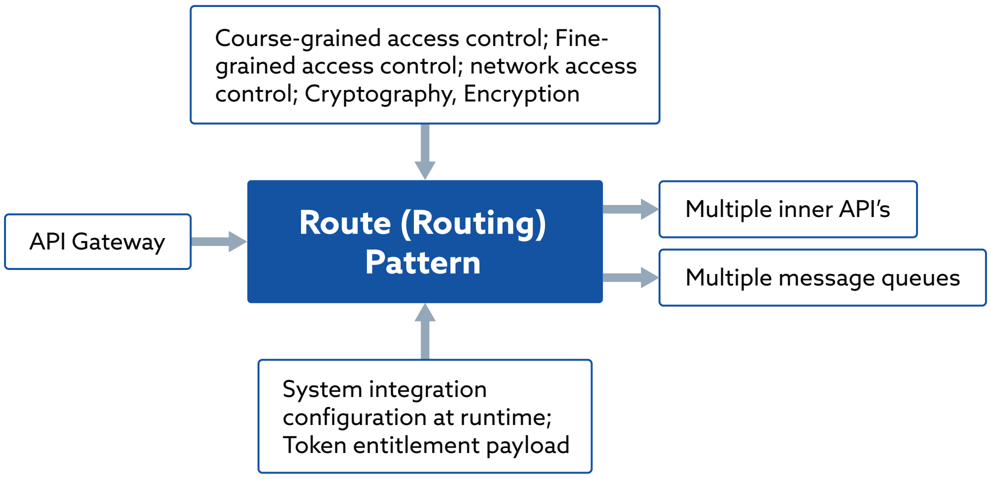

Encryption and Cryptography; Policy based traffic management
Enterprise Plane
Offload Pattern
Inter-mediation
Outer API and cluster ingress point
TLS end-point termination configuration; SAML/ Oath token update/ refresh mechanism

| Offload Pattern | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Demonstrate opportunities to consolidate technical capability |
| **Plane Location** | Enterprise (Platform) Plane |

| | |
|---|---|
| **Structural Description** | The API Gateway is the entry point of external traffic ingress into the microservice application |
| **Behavior Description** | Offload TLS certificate hosting; TLS Termination; AuthN and AuthX request brokering |
| **Data Disposition** | Data in transit |
| **Major Dependencies** | Platform Plane interconnection with IAM Platforms, Certificate Management Platforms; Upstream ingress bastion firewall; upstream load balancing platforms. |
| **Minor Dependencies** | Container fleet; Adjacent security logging interconnections |
| **Internal Event/ Messaging Needs** | HTTPS version 2/3; AS2 |
| **External Event/ Messaging Linkage** | API level logging; Gateway level syslog; AuthN and AuthX message exchange. |
| **Event Response Behavior** | Send, not receive. No transformation; Raw machine output |
| **Common Upstream Linkage** | Firewall; Global and/or local traffic load balancing; Internetwork ACL's |
| **Common Downstream Linkage** | Certificate authority; Configuration management; API Registry; Cluster management API security context; Machine ID/Service ID credential escrow. |
| **Ops Security Tie-Back** | API performance monitoring; syslog monitoring; machine health monitoring; anomaly detection |
| **DevSecOps Tie-Back** | Secure SDLC; API Registry; Swagger/YAML API definitions; AuthN and AuthX controls |
| **Evaluation Method** | API Performance trend analysis; SIEM log analysis and correlation with upstream events; anomaly correlation across API access ports and protocols. |
| **Control Overlay Disposition** | Inheritable from platform, not built into the API specification or container infrastructure |
| **Composite Status (unique/ generalized)** | Generalize; other patterns consume this overlay |

## 5.1.2 Route (Routing) Pattern

Use a route pattern when a single endpoint needs to expose multiple services behind it, and route requests and messages based on the incoming request. Routing policy and route traffic management are embedded in service mesh and / or using a sidecar service.   In the absence of route policy and traffic management embedded in service mesh, a combination of API specifications and route logic pertaining to message queue design is needed to enforce API topology (i.e. "who can talk to who, who can get messages from who"). Historically, large distributed monolithic legacy applications rely upon message queues to shuttle transaction information so an application can maintain state. If a service mesh is unavailable for use, a microservice may have to possess routing logic and perform utility functions that a sidecar service would perform. Alternatively, the routing pattern (or function) can sit on the API gateway. Route patterns can exist in hardware or software.

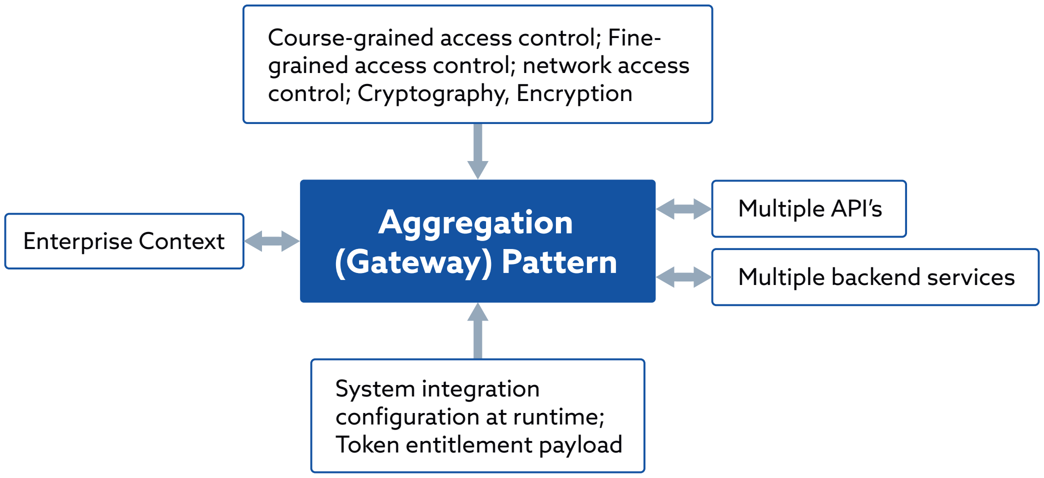

Course-grained access control; Fine-grained access control; network access control; Cryptography, Encryption
API Gateway
Route (Routing) Pattern
Multiple inner API's
Multiple message queues
System integration configuration at runtime; Token entitlement payload

| **Route Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Routing requests traffic to different versions or instances of the same service based on data within the request such as request headers that identify identity, source, client types, request host value, request URI path elements etc. An example of usage of the routing pattern is routing to a new version of a service for limited time based on the source of the request originating as determined by the source IP address.<br><br>Routing destinations could also be message queues instead of synchronous service endpoints. |
| **Plane Location** | Enterprise (Platform) Plane |

| | |
|---|---|
| **Structural Description** | Requests are routed to a specific version or instance of a service or message queue based on request  or message data as defined by the routing policy configuration. |
| **Behavior Description** | Incoming requests are evaluated using request data and the routing policy configuration to determine the target service instance destinations or message queues. |
| **Data Disposition** | Uses data/ transits data |
| **Major Dependencies** | Routing component such as API Gateway, configured service mesh functions, side car must be available |
| **Minor Dependencies** | Routed destinations or message queues must be available. |
| **Internal Event/ Messaging Needs** | Target service and message queue destination health and availability |
| **External Event/ Messaging Linkage** | Service mesh logging and telemetry |
| **Event Response Behavior** | Adaptive or fallback routing in reaction to target service unavailability |
| **Common Upstream Linkage** | IAM Platform, Gateway Platforms (API, Load Balancing, Policy Based Routing) |
| **Common Downstream Linkage** | Queue, Data Repository, Other Inner API |
| **Ops Security Tie-Back** | API Gateway availability<br>DDOS prevention<br>Scale or throttle traffic to ensure availability |
| **DevSecOps Tie-Back** | SAST - Service mesh configuration audit,<br>DAST - AuthN and AuthZ tests for service endpoints<br>IAST- Discard false positives |
| **Evaluation Method** | Observability, telemetry and logging of the routing function. Logging provides real-time visibility and observability. |
| **Control Overlay Disposition** | Preventive - DEVSECOPS controls, DDOS Prevention<br>Detective and Corrective - Scale/Throttle to ensure availability |
| **Composite Status (unique/ generalized)** | Generalized |

## 5.1.3 Aggregation Pattern

The Aggregation pattern receives and makes requests to multiple microservices, and then combines multiple requests to backend services into a single request to respond to the initial request. Software defined aggregation commonly occurs at the cloud edge when many devices are sending transaction messaging and activity logging to a central point of presence. Other patterns may come into play behind the gateway aggregation such as facade patterns, proxy patterns, and circuit breaker patterns depending upon the application goals and communication characteristics. These design patterns have a similar structure but the intent or purpose for which they are used is different.

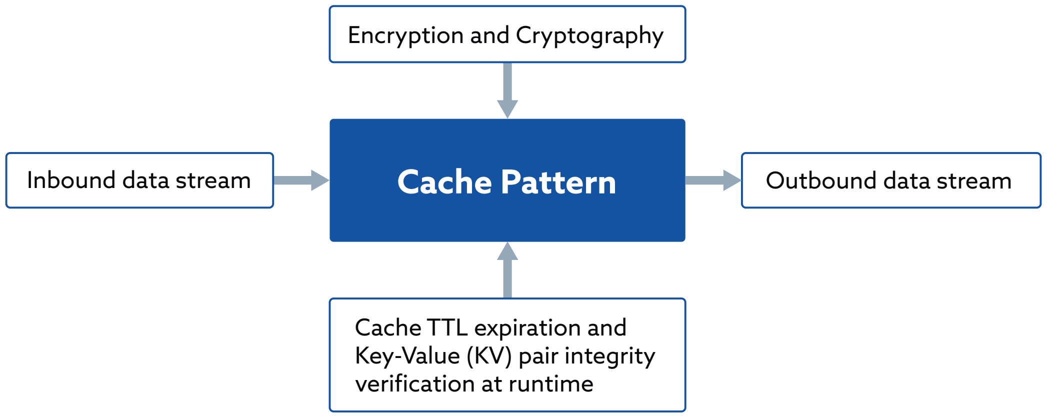

Course-grained access control; Fine-grained access control; network access control; Cryptography, Encryption
Enterprise Context
Aggregation (Gateway) Pattern
Multiple API's
Multiple backend services
System integration configuration at runtime; Token entitlement payload

| **Aggregation Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | The purpose of the Aggregation (gateway) pattern is to reduce the number of requests that the application makes to backend services and improve application performance over high-latency networks. |
| **Plane Location** | Enterprise (Platform) Plane |
| **Structural Description** | Aggregation (gateway) pattern is used to aggregate multiple individual requests into a single request or response. |
| **Behavior Description** | Aggregation (gateway) pattern is useful when a client is required to issue multiple calls to a variety of backend systems to carry out an operation. |
| **Data Disposition** | Data may need to be secured in-use if the aggregator is combining datasets from multiple endpoints. Otherwise, the aggregator may only be transmitting data between requestors and responders. |

| | |
|---|---|
| **Major Dependencies** | Networks (Networks could introduce significant latency) |
| **Minor Dependencies** | Availability & proximity (Availability of various systems that this pattern will communicate with) |
| **Internal Event/ Messaging Needs** | Aggregator pattern securely authenticates the request and passes an access token. Services can verify that the requestor is authorized to perform the operation(s) |
| **External Event/ Messaging Linkage** | Network security controls (Network Access Controls- AuthN, Authz, protecting data at rest etc. using encryption) |
| **Event Response Behavior** | Ensure the Aggregation gateway has a resilient design to meet your application's availability requirements.<br>The Aggregation gateway may be a Single Point of Failure (SPOF) so it should be well equipped to handle load balancing.<br>Aggregation Gateway should have appropriate controls to ensure that any delay in response from the backend systems doesn't cause performance issues. |
| **Common Upstream Linkage** | Configuration management; API security; policy management and enforcement |
| **Common Downstream Linkage** | Maintenance, uptime/downtime, integrations, testing, vulnerability scans and patching |
| **Ops Security Tie-Back** | API performance monitoring; syslog monitoring; health monitoring; anomaly detection, Incident management, anti-malware/virus, vulnerability mitigation, patches |
| **DevSecOps Tie-Back** | Secure SDLC; Agile, simpler/flexible development and testing cycle, Continuous Integration / Continuous Delivery (CI/CD) pipeline |
| **Evaluation Method** | Logging/monitoring; alerting, metering alerts based on authorization failure conditions, SIEM Incident and Event Management |
| **Control Overlay Disposition** | Gateway improves and directly impacts the performance and scale of the application.<br>Preventative: vulnerability mitigation, patching<br>Corrective: Incident response<br>Detective: Performance monitoring, health monitoring |
| **Composite Status (unique/ generalized)** | Generalized |

### 5.1.4 Cache Pattern

Caching in application design regularly addresses requirements for increasing availability, improving application performance, and/or reducing backend data reading and writing. Caching can be embedded or distributed. Derivatives of the main caching patterns exist. If a business operation has ongoing value for later use, consider caching the result.  Examples include data elements that are priced per transaction such as a metered external interface. A second example is session authorization when the consumer interacts with many outer API's in a modern distributed microservice application. To simplify debugging, cache in the same layer in the same place to avoid multiple cache instances falling out of sync with each other.

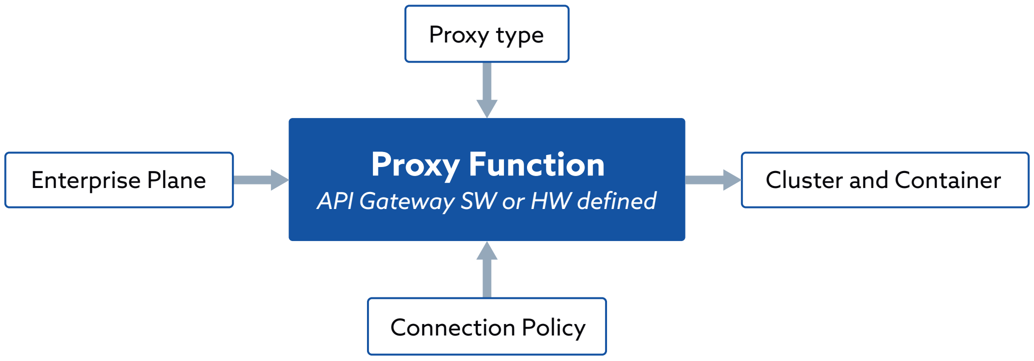

Encryption and Cryptography
Inbound data stream
Cache Pattern
Outbound data stream
Cache TTL expiration and Key-Value (KV) pair integrity verification at runtime

| **Cache Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Improve performance, scalability, and availability |
| **Plane Location** | Software Plane |
| **Structural Description** | Temporarily copies frequently accessed data to fast storage that is closer to the application |
| **Behavior Description** | Improve performance using rapid data delivery patterns in case of significant I/O data connectivity, improve availability using caching of credentials when application session interruption occurs. By avoiding latency due to fast retrieval of data, caching  eliminates the need to scale frequently. |
| **Data Disposition** | Data in use (memory) |

| | |
|---|---|
| **Major Dependencies** | Eventual consistency, cache size, eviction policy based on cache hit ratio or least-recently-used policy, cold starts, caching fleet outages, changes in traffic patterns,  extended downstream outages, Network latency, Request Coalescing |
| **Minor Dependencies** | Type of communication protocols, caching service management |
| **Internal Event/ Messaging Needs** | Inner API, SOAP, RPC, ICP, HTTP/S |
| **External Event/ Messaging Linkage** | AppDev Debug (Stricter based on compliance requirements) |
| **Event Response Behavior** | Send/Receive/Queuing/Messaging, Event Driven, Asynchronous/ Synchronous |
| **Common Upstream Linkage** | Global and/or local traffic load balancing |
| **Common Downstream Linkage** | Inner API; Port and protocol tightly coupled to the data source of record, or data source of reference. |
| **Ops Security Tie-Back** | API monitoring; anomaly detection; Security Monitoring Process |
| **DevSecOps Tie-Back** | SAST for Caching size/memory overrun issues, DAST, cache client library vulnerability patching |
| **Evaluation Method** | Hardening guide for API (Hardening guide in policy as code form if containers are in use), Network latency and performance, Load testing, Testing methodologies for cache invalidation and deception |
| **Control Overlay Disposition** | API Tracing/Debug, cryptographic overlay (Data in transit, process and use), Spin up new cache instances (shorter TTL, dynamically scaling memory configurations) to avoid cache-miss, cache-fail and load balancing. |
| **Composite Status (unique/ generalized)** | Generalized; other patterns consume this overlay |

### 5.1.5 Proxy

A proxy pattern is a hardware enabled, or software-defined, construct which intermediates machine to machine readable data streams.  Proxy capability manifests in different design orientations.  A proxy 'speaks' or 'acts' on another's behalf either to conceal the requestor, or shoulder delegated responsibility granted by the requestor.   In machine-to-machine data transmissions a proxy can be transparent, whereby the proxy function intermediates between the requester and the content provider, or intermediates between an initiating API endpoint, and responding API endpoint. Transparent proxy functionality intercepts data and performs actions such as caching, offloading, or redirection, but it does not modify the data stream (modification of the data stream would be the responsibility of the Adapter pattern.) Reverse proxy functionality responds on the provider's behalf, offering concealment.  Other types of proxies exist.  Distributed microservice applications commonly utilize Transparent and Reverse proxies.



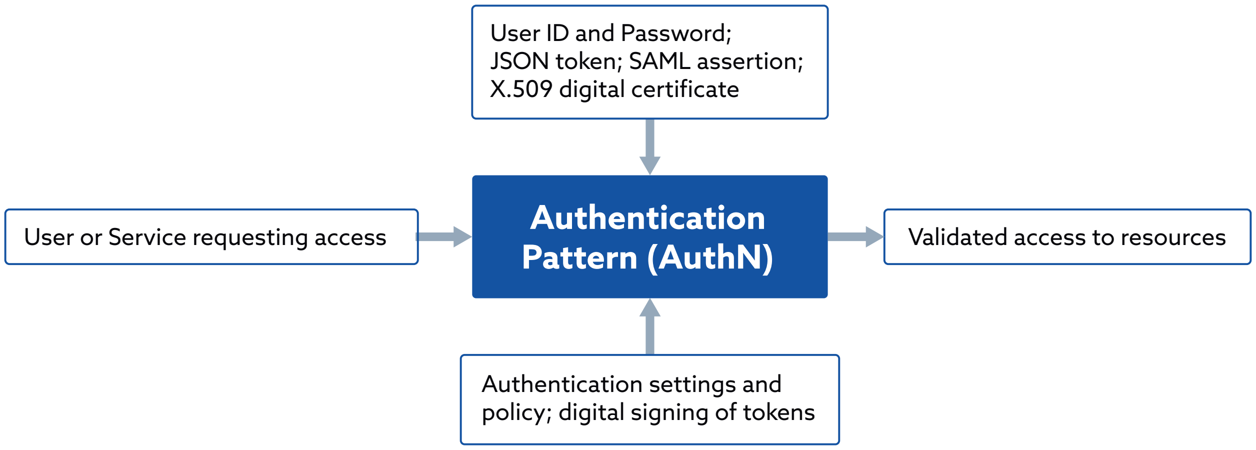
| **Proxy Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Brokers connections between service consumer and service provider |
| **Plane Location** | Software Plane |
| **Structural Description** | Proxy accepts input from various service consumers, acting as a security checkpoint to the API or container.  The input from the service consumer is then processed against a security policy, with the outcome either accepting, rejecting, or dropping connections to the service provider. |

| | |
|---|---|
| **Behavior Description** | Proxy patterns may be extended or specialized to include a network or access control proxy (i.e., focused on network layer three or layer four attributes of source, target, port, and protocol), an identity proxy (i.e., enforce authentication and authorization), and so on.  Proxy may also collaborate with other patterns (e.g., add meta-data to the incoming request to help facilitate appropriate routing.)  Datastream input which is received is evaluated against a security policy.  Assuming there is no policy violation, the request proceeds from the service consumer to the service provider. |
| **Data Disposition** | Proxy is expected to pass communication from the service consumer to the service provider.  It is not expected to store data, with a possible exception of requests being temporarily queued for evaluation.  Proxy may, but is not required, to enrich the data stream with meta-data if so desired. |
| **Major Dependencies** | Proxy expects that network connectivity, runtime environment, and application endpoints are functional and responsive. |
| **Minor Dependencies** | Proxy is expected to continue enforcing security policy, even if disconnected from the policy definition component.  Note that the policy enforced would potentially be an outdated/point-in-time version when unable to get the most recent version of the security policy. |
| **Internal Event/ Messaging Needs** | A nominally functioning Proxy can operate independently, but may benefit in interactions with the following patterns:<br><br>AuthN - check that authentication credentials are still valid, and have not been revoked.<br><br>Offload - take a decrypted TLS stream and inspect for policy compliance.<br><br>Route/Routing - ensure that service consumer meta-data (e.g., original source IP address) is conveyed to the Route/Routing pattern. |
| **External Event/ Messaging Linkage** | Proxy should be capable of providing event details and decisions to a security analytics solution (e.g., SIEM) for the purposes of enterprise security event correlation and incident investigation.<br><br>Proxy would ideally be capable of ingesting threat intelligence or high-confidence lists of suspect or compromised users, devices, and endpoints, which may inform security policy compliance decisions. |

| | |
|---|---|
| **Event Response Behavior** | Proxy response follows one of three options:<br><br>ACCEPT, which allows the datastream or connection from service consumer to service provider, and optionally provides a response to the consumer.<br><br>DROP, which prevents the connection, and does not provide a response to the consumer.<br><br>REJECT, which prevents the connection, and is expected to provide a response to the consumer. |
| **Common Upstream Linkage** | AuthN, Offload |
| **Common Downstream Linkage** | Route/ Routing |
| **Ops Security Tie-Back** | Infrastructure as Code (IaC), configuration management, Security Information Event Management (SIEM), load balancer, firewall |
| **DevSecOps Tie-Back** | Continuous Integration / Continuous Delivery (CI/CD) pipeline for consistent and repeatable implementation, Static Application Security Testing (SAST) for secure implementation, Dynamic Application Security Testing (DAST) for policy enforcement |
| **Evaluation Method** | Analysis of communications allowed and denied, correlation with enterprise data to confirm proxy is allowing/blocking as expected, and active attempts to breach the proxy (e.g., red tea/pen test) |
| **Control Overlay Disposition** | Preventive/Corrective - access control<br>Detective - adds perspective to enterprise correlation. May aid incident investigation<br>Regulatory - may be required to satisfy auditors or corporate customers |
| **Composite Status (unique/ generalized)** | Generalized, Has unique control overlay<br><br>Proxy control overlay is likely generalized and similar to other patterns. |

### 5.1.6 AuthN (Authentication) Pattern

The AuthN (authentication) Pattern ensures services and users that access microservices prove that they are who they say they are. For example, OpenID Connect extends the authentication procedures of OAuth 2.0. OAuth 2.0 is an authorization framework incorporating authentication procedures. An identity token is given to the relying party from the authentication server and contains claims about the authentication and identity information (usually after signing in through a portal).


User ID and Password;
JSON token; SAML assertion;
X.509 digital certificate
User or Service requesting access
Authentication Pattern (AuthN)
Validated access to resources
Authentication settings and policy; digital signing of tokens

| **AuthN Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Process to establish trust and verify the identity of the entity he/she/it claims to be |
| **Plane Location** | Software and Enterprise (Platform) plane |
| **Structural Description** | Client - whether user or other microservice - needs to be identified before consuming service |
| **Behavior Description** | Authenticates the user or service before the authorization pattern determines access to resources. Authentication consists of two related concepts, "identification" of the user or service, and "verification" that the user or service is in fact who or what they claim to be. |
| **Data Disposition** | The authentication pattern may generate a boolean response as to whether or not the user or service has verified its identity. Other responses may include a token or assertion that the entity is who they claim to be. API endpoints, API gateways, MESH and ISTIO are consumers of identity tokens or assertions. |

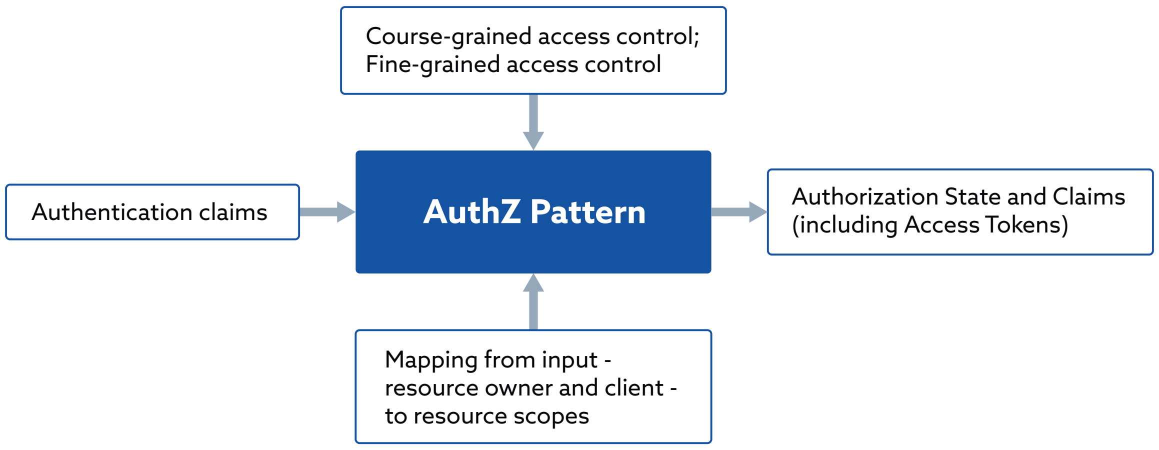
| | |
|---|---|
| **Major Dependencies** | Authentication relies on a trusted identity repository where credentials are stored (e.g., Active Directory, LDAP), and may also depend on Public Key Infrastructure (PKI) for signing identity assertions or tokens and verification of those tokens or assertions. |
| **Minor Dependencies** | Depending on the authentication protocol used, there may be a client (relying party), user (Resource owner), and authorization server that issues access tokens (IDP) |
| **Internal Event/ Messaging Needs** | Token and client event messages for OAUTH, OAUTH policies that trigger alerts – investigate and control, logs from istio components, AuthN and AuthZ message exchange. |
| **External Event/ Messaging Linkage** | AppDev Debug, SIEM |
| **Event Response Behavior** | During policy enforcement, respond with allowed or denied.  Be prepared to log authentication events for enterprise event correlation. May be required to generate an alert if too many authentication events for a specific user exceed a certain threshold (e.g., more than three authentication failures in 60 seconds.) |
| **Common Upstream Linkage** | Firewall; Global and/or local traffic load balancing; Internetwork ACL's |
| **Common Downstream Linkage** | Certificate authority; Configuration management; API Registry; Cluster management API security context; Machine ID/Service ID credential escrow, Proxy, Authorization (AuthZ). |
| **Ops Security Tie-Back** | API performance monitoring; syslog monitoring; machine health monitoring; anomaly detection |
| **DevSecOps Tie-Back** | Secure SDLC; API Registry; Swagger/YAML API definitions; AuthN and AuthX controls |
| **Evaluation Method** | API Performance trend analysis; SIEM log analysis and correlation for anomaly detection and incident response. |
| **Control Overlay Disposition** | Preventative – Access control |
| **Composite Status (unique/ generalized)** | Generalize; other patterns consume this overlay |

### 5.1.7 AuthZ (Authorization) Pattern

AuthZ (authorization) determines what an authenticated user can do or what permissions they have. A standard way of passing these permissions from the authorization server to the resource servers is by encoding scopes into a JWT. Auth Service determines the permissions based on the user's role/attribute. A summary of the steps is provided:

- After the AuthN procedure, the client sends an authorization grant request to the AuthZ service along with the authorization token from the AuthN service, the request is passed on to the AuthZ service.
- The AuthZ service verifies the authorization token and returns an access code usually in the format of a JWT with "permissions" as requested by the user/role.
- The request after the AuthZ procedure to the API Gateway includes the JWT in an "Authorization header".

AuthZ is for a client to request authorization or permissions to resources. The client sends the AuthZ service user authentication result in the form of an authorization code, and if successful, the AuthZ service responds with an access code typically embedding the scope information. The scope shall be mapped to the details of permitted actions on the resources by the resource servers. An access token is typically encoded in the format of JSON Web Token (JWT).


Course-grained access control; Fine-grained access control
Authentication claims
AuthZ Pattern
Authorization State and Claims (including Access Tokens)
Mapping from input - resource owner and client - to resource scopes

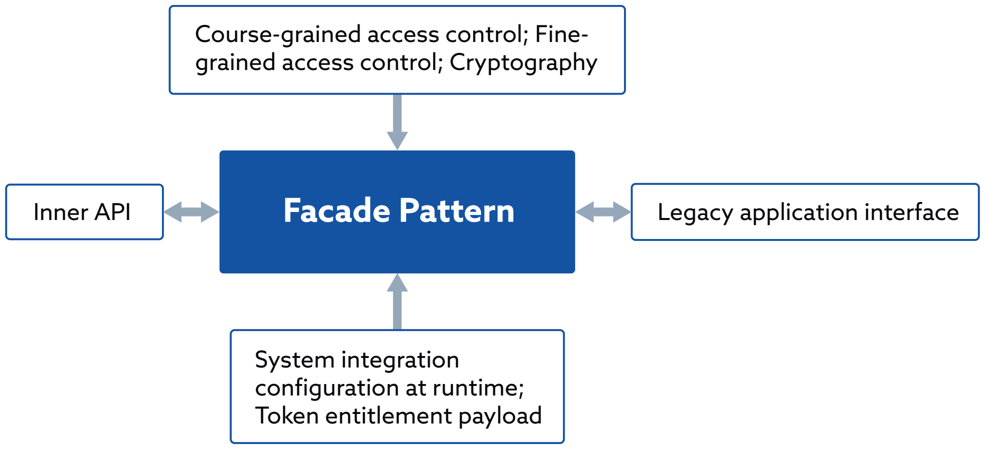
| AuthZ Pattern | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Determining what the client is authorized to do on behalf of the user based on the authorization policies configured in the AuthZ service. |
| **Plane Location** | Software and Enterprise (Platform) plane |

| | |
|---|---|
| **Structural Description** | AuthZ mechanism returns information that will be used to determine if the user can perform based on the request they have made. |
| **Behavior Description** | The AuthZ service receives resource owner's authentication claims via the client, usually in the form of an authorization code, and maps the resource owner and the client to the authorized resources usually in the form of scopes in an access token.<br><br>Note: If the Authorization fails, then the request should be dropped. The requestor would need to issue a new request again for re-authorization. |
| **Data Disposition** | Username, clientID, authorization token and possible scope. |
| **Major Dependencies** | AuthN service, Access/JWT token |
| **Minor Dependencies** | HTTP POST/GET, TLS, Certs, Access Control |
| **Internal Event/ Messaging Needs** | Validate the input including the client credential and authorization code. Look up the authorization policies to match resource owners and clients to authorized resources in terms of scopes. |
| **External Event/ Messaging Linkage** | Input validation and authorization policy lookup results may be logged for security monitoring and audit. |
| **Event Response Behavior** | If the input validation and authorization lookup are successful, an access token and authorized scope are returned with HTTP status code of 200.<br><br>If failed, one of the following HTTP status codes would ideally be returned:<br><br>HTTP 401: Unauthorized.  A problem has occurred relating to authentication.<br><br>HTTP 403: Forbidden.  Client denied access to a resource.<br><br>Other status codes may optionally be provided to the client. |
| **Common Upstream Linkage** | Authentication service, Upstream microservice (service to service), Edge service (routing, security), offloading |
| **Common Downstream Linkage** | Downstream microservice (service to service), API gateway, Internal services, integrations, Facade pattern, Aggregation pattern |

|  |  |
|---|---|
| **Ops Security Tie-Back** | API performance monitoring; syslog monitoring; health monitoring; anomaly detection, Incident management, Anti-malware/virus, Vulnerability mitigation, Patches, testing, Maintenance, uptime/ downtime |
| **DevSecOps Tie-Back** | Secure SDLC; Agile, simpler/flexible development and testing cycle, smaller changes, rapid development (Continuous Integration/ Continuous Delivery) |
| **Evaluation Method** | Logging/monitoring; alerts based on authorization failure conditions, SIEM Incident and Event Management |
| **Control Overlay Disposition** | Access controls are Preventive controls |
| **Composite Status (unique/ generalized)** | Generalized, other patterns consume this |

### 5.1.8 Facade Pattern

In the canonical software architecture and design paradigm, a facade is a structural pattern serving as a front-end interface to more complex software coding underneath. Facades can mask the interactions of complex components and serve as a simplified means for client applications and interfaces to make delegated calls to more complex software interactions. Facades can add layers of depth to structure software subsystems into layers (Facades that access other additional lower level facades). When decomposing monoliths, facades might not be the best choice to front newly developed microservices. Long-term it is better suited to use a proxy pattern as it can be dispensed with when no longer needed and would be easier to implement compared to the Facade pattern.

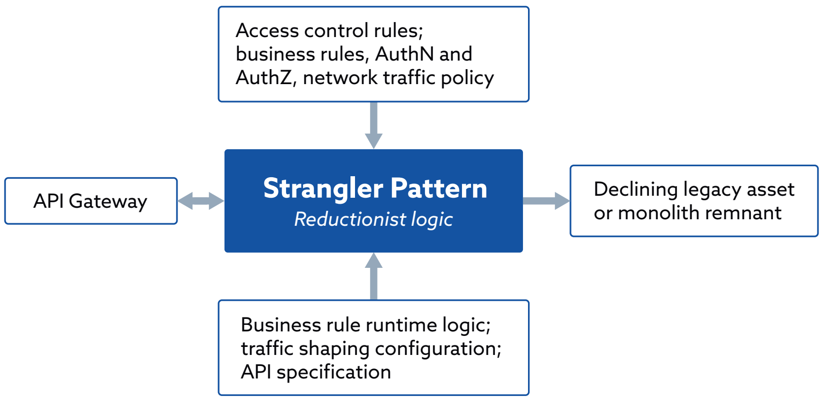

Course-grained access control; Fine-grained access control; Cryptography
Inner API
Facade Pattern
Legacy application interface
System integration configuration at runtime; Token entitlement payload

| Facade Pattern | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | The purpose of the Façade pattern is to allow for protocol mediation between the Client and Microservices to provide improved user experience. Clients do not have to know about Microservices, just the API and secure configuration from the Façade. Client access can be simpler by using coarse grain APIs provided by the Facade. |
| **Plane Location** | Enterprise (Platform) Plane |
| **Structural Description** | The security context Facade Pattern serves as the API ingress/egress point to underlying microservices supporting an API. The security context for the software plane is in the API conversion from the single Facade interface (Inner API) into the many different microservice API's (Outer API). |
| **Behavior Description** | The Facade pattern is a simple way for the clients to interact with the subsystems and make changes to the subsystem classes without affecting the client code. Instead of being tightly coupled, clients are now loosely coupled with the subsystem classes. |
| **Data Disposition** | Consider security for the services, communication between services, protection of data in transit & in use |
| **Major Dependencies** | Specific REST APIs (There is a mutual dependency of the Facade with Outer APIs and Inner APIs.),<br><br>Protocols: (Facade needs to understand protocols for both client and microservice really well since it provide protocol mediation)<br><br>Availability of various subsystems: Facade can be a Single Point of Failure as otherwise clients cannot communicate with the microservices |
| **Minor Dependencies** | Design changes in the subsystems<br>Connectivity/AuthN for various subsystems<br>AuthZ (XACML, RBAC, ABAC) will establish privileges required by the clients for various subsystems |
| **Internal Event/ Messaging Needs** | Telemetry, logging and observability information passed to Internal event processing or messaging. |
| **External Event/ Messaging Linkage** | Instead of clients being tightly coupled with the sub systems, Facade provides an interface to the clients by exposing the appropriate APIs to the clients. So, clients delegate the Facade to perform actions in their stead. |

| | |
|---|---|
| **Event Response Behavior** | It would be preferable to invoke the circuit breaker pattern on degraded performance. |
| **Common Upstream Linkage** | Microservices (Outer API) Configuration management; API security; policy management and enforcement, circuit breaker. |
| **Common Downstream Linkage** | API Client and API gateway (Inner API) Configuration management; API security; policy management and enforcement. |
| **Ops Security Tie-Back** | IT Security controls, integration with Infrastructure-as-Code.API performance monitoring; syslog monitoring; health monitoring; anomaly detection, Incident management, Anti-malware/virus, Vulnerability mitigation, Patches. |
| **DevSecOps Tie-Back** | CI/CD pipeline integration with code security tooling, ex: snyk, stackhawk, etc. Security testing tools such as SAST, DAST, 3rd party vulnerability scanning tool (Additional tooling examples include Twistlock and PureSec (PANW). |
| **Evaluation Method** | Verification of controls. Logging/monitoring; alerting, metering alerts based on authorization failure conditions, SIEM Incident and Event Management. |
| **Control Overlay Disposition** | Prevent: SAST, DAST, vulnerability scanning<br>Detect: Performance monitoring, health monitoring, anomaly detection. |
| **Composite Status (unique/ generalized)** | Generalized, uses the same control overlay as aggregation pattern |

### 5.1.9 Strangler Fig Pattern

A strangler pattern 'chokes off' legacy application functionality incrementally. In the presence of a strangler pattern, an older application surrenders specific functionality to the new modern code base. Eventually, the new code base replaces the older system's functionality and the legacy system retires from service. Using the pattern as an "old to new" transitional approach to breaking up a monolithic application, intermediate software architecture states make use of other patterns such as proxy and wrapper patterns so that new microservices can interact with the remaining legacy code base. Eventually the monolith breaks apart and deflates in a controlled manner, and the wrapper and proxy patterns can also retire. This pattern's work factor has to be noted.


Access control rules; business rules, AuthN and AuthZ, network traffic policy
API Gateway
Strangler Pattern
Reductionist logic
Declining legacy asset or monolith remnant
Business rule runtime logic; traffic shaping configuration; API specification

| Strangler Pattern | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Purpose of this pattern is a way to gradually/incrementally migrate a legacy backend system to new architecture. Application monoliths can be replaced wholesale (greenfield deployment) or "evergreened" as certain functions are carved out of the monolith and updated such that they can function independently from the monolith's processing boundary. It is recommended to pair this pattern implementation with the blue-green deployment to unhold Business Continuity Plans. |
| **Plane Location** | Software Plane |
| **Structural Description** | Enables incrementally replacing specific pieces of functionality with new applications or services. This creates two separate applications that live side by side in the same URI space. Over time, the newly refactored application "strangles" or replaces the original application in phases until finally you can shut off the monolithic application. Each migration phase will have a router or logic that determines which service(legacy or new) to route to for specific functionality |
| **Behavior Description** | Parallel run of legacy and microservice pattern until the legacy monolith is completely replaced. Each of the phases will have a router or logic that determines which path or route is taken for specific legacy or new functionality. Strangler also depends on Event Driven Architecture patterns[9] to manage the state and data across legacy monolith components and microservice(s). New microservice domain models may need to include legacy domain attributes. The state and data is maintained by bi-directional sync with translation if needed. Clients will have to plan for the Eventual Consistency https://ibm-cloud-architecture.github.io/refarch-eda/patterns/intro/ |

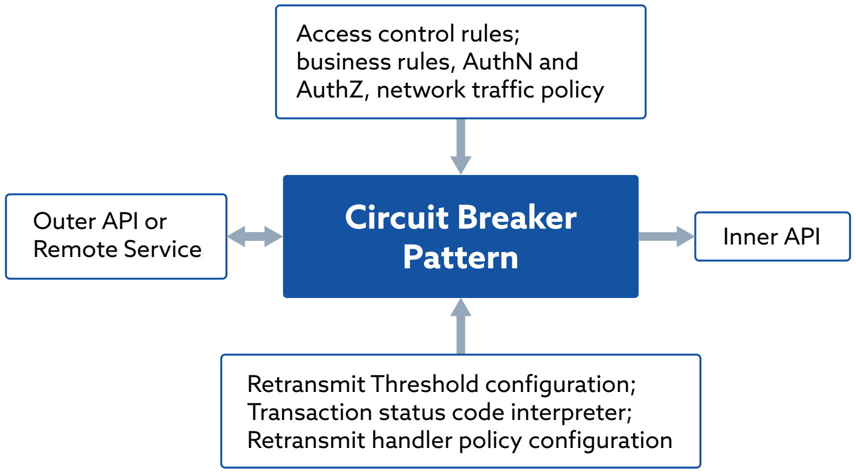
[9] IBM. Patterns in Event-Driven Architectures – Introduction. IBM Garage Event-Driven Reference Architecture. Retrieved August 11, 2021, from https://ibm-cloud-architecture.github.io/refarch-eda/patterns/intro

| | |
|---|---|
| **Data Disposition** | Transit data/ Uses data |
| **Major Dependencies** | Requests to the backend system can be intercepted. Single point of failure of façade and performance bottlenecks. Eventual consistency is a trade-off with REST JSON, especially when crossing multiple security zones or application layers. If in the design the requirement is a hard commit then consistency has to be closed or mitigated with other capabilities like caching, and lazy writes tied to user session. |
| **Minor Dependencies** | Access to backend data sources to both new and legacy monolith |
| **Internal Event/ Messaging Needs** | Inter-process and  Inter-pattern.  Depending on the application or system context, the messaging patterns described in this source[10] can be leveraged to Strangler applications to provide for data and application state sync. |
| **External Event/ Messaging Linkage** | AppDev and database transaction logging. |
| **Event Response Behavior** | Send/Receive/Queuing/Messaging |
| **Common Upstream Linkage** | Façade, API Gateway, API Proxy |
| **Common Downstream Linkage** | Service Bus, API proxy, Event Adapter, Service Bus Queue/Topic, Adapter, Sagas |
| **Ops Security Tie-Back** | IaC - Infrastructure as a Code; Policy as Code as driven by the DEVOPS pipeline (typically Jenkins but can be others) |
| **DevSecOps Tie-Back** | Code quality review pre-unit test; 95% testing coverage; pipeline driven code quality SAST during unit testing at repository level (pull request sign-off); pipelined DAST in system integration testing as build requirement; pipelined automated deploy to container environment following clean API topology as per API (swagger or YAML) specification. |
| **Evaluation Method** | Pipelined automated testing according to pass/fail infrastructure policy as code; API performance, load testing |
| **Control Overlay Disposition** | Detective |
| **Composite Status (unique/ generalized)** | Generalized, Has unique control overlay  -  generalized |

[10] Enterprise Integration Patterns. Enterprise Integration Patterns - Messaging Patterns Overview. Retrieved August 11, 2021, from https://www.enterpriseintegrationpatterns.com/patterns/messaging/.

### 5.1.10 Circuit Breaker Pattern

A circuit breaker pattern is a means to stop or limit request-response interaction to prevent a larger failure should a service enter a threatening state or stop working. The pattern is a means to prevent failures from growing bigger or failing completely (i.e., closed_state) and resides in the software plane. Threatening signs can be the data exchange surpassing defined thresholds or consecutive failures surpassing defined thresholds. The result can be that the connection "trips" and goes quiet for the duration of the timeout period (i.e., open_state) and returns an error to the requestor; after the timeout expires the pattern may open the connection up again; or alternatively the result can be that allowing a few transactions to pass successfully (i.e., half_open). If it fails a second time, a second time out period ensues, until the microservice returns to closed_state.


Access control rules; business rules, AuthN and AuthZ, network traffic policy
Outer API or Remote Service
Circuit Breaker Pattern
Inner API
Retransmit Threshold configuration; Transaction status code interpreter; Retransmit handler policy configuration

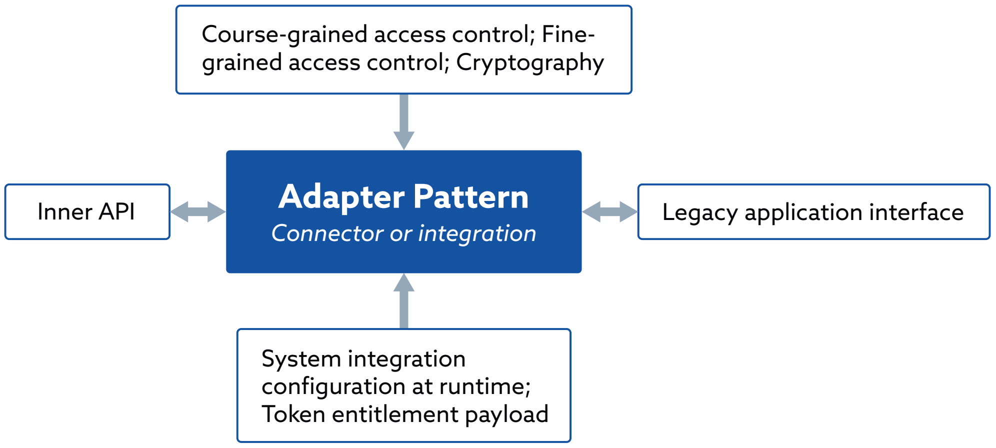
| **Circuit Breaker Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | To build a fault tolerant and resilient system that can survive gracefully when downstream services are unresponsive due to resource issues such as network or service failures. |
| **Plane Location** | Software Plane |
| **Structural Description** | It monitors downstream service request-response interactions, and detects failures. It protects upstream services from cascade failures and responds with default response/error or the last result from cache. It checks downstream service state whether it is recovered from failure state. |

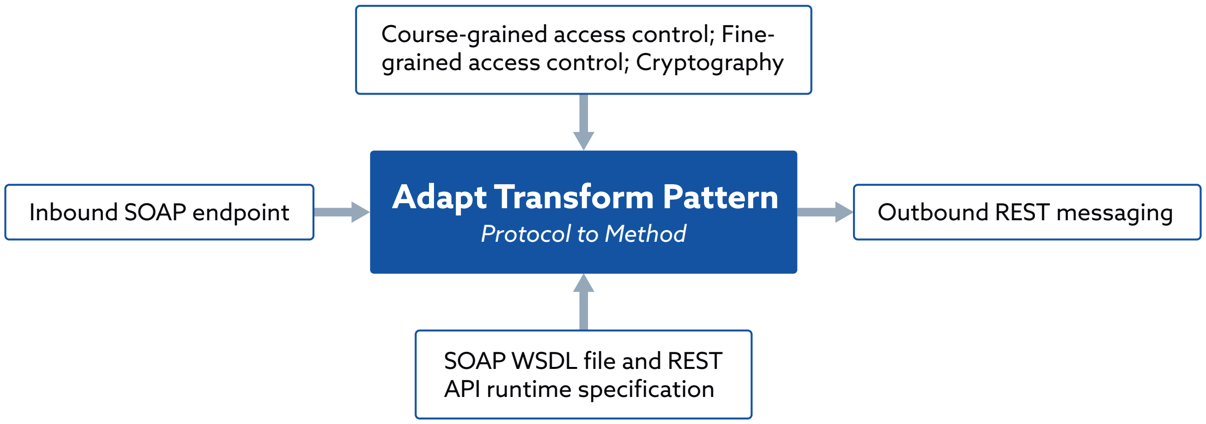
| | |
|---|---|
| **Behavior Description** | Circuit breaker pattern allows request-response interaction between upstream and downstream services when it is in closed_state. Circuit breaker trips the connection and changes the state to open_state for the duration of timeout period when it observes downstream call failures more than threshold value. Circuit breaker allows the interactions to check the downstream service connectivity or resource availability after specified time period, i.e., in a half_open state. If a downstream service allows interactions successfully, the circuit breaker closes the connection and allows request-response interaction between upstream and downstream services.<br>1. It monitors downstream service requests and responses, and detects failures.<br>2. It stops cascading some or all requests to downstream services.<br>3. It responds to the requests with either standard responses or error conditions. |
| **Data Disposition** | Transits data/ Uses data |
| **Major Dependencies** | Optimal threshold configuration, Network connectivity, Services availability. |
| **Minor Dependencies** | API Gateway, Threshold configuration |
| **Internal Event/ Messaging Needs** | It works independently however it can leverage the Retry pattern to resume interactions to downstream services and Cache pattern to store and serve the last result to upstream services when circuit break is in open state. |
| **External Event/ Messaging Linkage** | The activity needs to be logged for debugging and also triggers the event notifications to the Operations team to take the appropriate actions on downstream service or network failures. |
| **Event Response Behavior** | This pattern responds with standard error or last result to upstream services when the circuit breaker opens up the connection.<br>Returns HTTP status code 503 when the downstream service is unavailable. |
| **Common Upstream Linkage** | Proxy, Route |
| **Common Downstream Linkage** | Retry, Cache |
| **Ops Security Tie-Back** | Traffic monitoring, Performance monitoring, Services availability, Configuration management. |

|  |  |
|---|---|
| **DevSecOps Tie-Back** | Continuous Integration / Continuous Delivery (CI/CD) pipeline, Status Application Security Testing (SAST), Dynamic Application Security Testing (DAST) |
| **Evaluation Method** | Analyse the behaviour with various threshold configurations and simulation of downstream service failures. Logging and telemetry analysis. |
| **Control Overlay Disposition** | Preventive, Corrective, Detective, Mitigate |
| **Composite Status (unique/ generalized)** | Generalized |

### 5.1.11 Adapter (wrapper/translate/transform) Pattern

An adapter pattern converts a data stream into a representation that an otherwise incompatible interface could not interpret. The requesting client sees only the target interface, and not the adapter that sits behind it.  As a structural pattern, adapter patterns create bridges between two independent unconnectable interfaces such that the pattern "wraps" an incoming request so that the receiving interface can make sense of the request. Adapter patterns wrap incompatible protocols to facilitate inter-communication, or wrap data formats to aid dataframe conversions from one format to another (i.e. Vendor 3rd party data connectors or adapters). If the adapter pattern 'decorates' the code, it adds functionality at runtime without modifying the underlying structure (i.e. a wrapper class to avoid creating additional subclasses in software).


Course-grained access control; Fine-grained access control; Cryptography
Inner API
Adapter Pattern
Connector or integration
Legacy application interface
System integration configuration at runtime; Token entitlement payload


Course-grained access control; Fine-grained access control; Cryptography
Inbound SOAP endpoint
Adapt Transform Pattern
Protocol to Method
Outbound REST messaging
SOAP WSDL file and REST API runtime specification

| **Adapter Pattern** | |
|---|---|
| **Version** | 1.0 |
| **Pattern Purpose** | Provides an interface to the Data Conversion / Transformation capabilities (E.g..: translating data from one format to another within the same application or a migration of data from one system to another). Provides an interface to Migration capabilities, where the transfer of data to another system may result in the repository change (at a service or schema mapping level). |
| **Plane Location** | Software Plane |
| **Structural Description** | Transform may be explicit or implicit (automatic) data conversion. The transform pattern can be part of a system where the application layer converts the data type to be put in the database table, or serve as a means to transform one API communication protocol to another. |
| **Behavior Description** | Each programming language has its own rules on conversion. The intention of the pattern is to transform a data type or communication protocol or method. When transforming to restful methods, expect eventual consistency in transactions and messages. |
| **Data Disposition** | Data in transit and data at rest |
| **Major Dependencies** | Platform Plane interconnection with IAM Platforms, Certificate Management Platforms; Internal core network; Message Queue Management Platforms, Upstream ingress bastion firewall; upstream load balancing platforms. Other inner API's |
| **Minor Dependencies** | Breakouts to different database-linked services needed, ACID (Atomic, Consistent, Isolated, Durable) not enforced. |

| | |
|---|---|
| **Internal Event/ Messaging Needs** | HTTPS version 2/3; AS2. Messaging may be in a legacy messaging format (SOAP), Private API's, CRUD RPC calls, client-server connectors using URIs and HTTP – REST, pub/sub messaging, gRPC |
| **External Event/ Messaging Linkage** | API level logging; Gateway level syslog; Authentication and Authorization of message exchanges such as RPC, gRPC, pub/sub messaging and Public API's. |
| **Event Response Behavior** | Event driven messaging, publishes events consumed by subscribers, asynchronous, loose coupling. |
| **Common Upstream Linkage** | Bastion Firewall; Global and/or local traffic load balancing; Internetwork ACL's |
| **Common Downstream Linkage** | Certificate authority; Configuration management; API Registry; Cluster management API security context; Machine ID/Service ID credential escrow. |
| **Ops Security Tie-Back** | API performance monitoring; syslog monitoring; machine health monitoring; anomaly detection |
| **DevSecOps Tie-Back** | Secure SDLC; API Registry; Swagger/YAML API definitions; Authentication and Authorization controls, Controlling and throttling API invocations and static code analysis. |
| **Evaluation Method** | API Performance trend analysis; SIEM log analysis and correlation with upstream events; anomaly correlation across API access ports and protocols. Pub/Sub messaging system that supports stream and event queues, as well as internal and external auditing. |
| **Control Overlay Disposition** | Rebuild and redesign boundaries not sharing tables across microservices. For transformation pattern explicit or implicit data conversions on one table. Communicate events across infrastructure. Implement API gateway that does handoff to transform patterns. During migration there are controls that are inheritable from the platform. |
| **Composite Status (unique/ generalized)** | Generalized |

# 6.0 Security Control Overlays

As stated previously, software design patterns guide software development. Security control overlays, a discrete collection that represents a fully specified set of controls, control enhancements, and supplemental guidance, integrates into the architectural design process as embedded and upfront administrative, technical, or physical requirements. Together, software design patterns and security control overlays inform software development to "build in" security as a design element, and not as an application or facade applied at the end where changes to software are most expensive to make. In the design of software, specifically microservices in this case, the opportunity manifested by applying security control overlays prior to any software being developed is to embrace software security as a *complete thought in an architectural sense.* The National Institutes of Standards Special Publication NIST SP 800-53 Revision 5 offers specific security architecture control guidance in the SA Control Family (Developer Security and Privacy Architecture and Design) and PL-8 (Security and Privacy Architectures) control objective. The aspirational goal of this paper is to avoid bringing security architecture and design to fully developed code but provide a means to have software security control conversations advance at the same pace of development where control capability does not exist beforehand.

In 2001, under the University of Maryland (Experimental Software Engineering Group (ESEG)) Papers, Boehm and Basili released findings outlining actions and outcomes that increase defects in developed software code. Both concluded that finding and fixing software problems are 100 times more expensive post-production than finding and fixing them during design[11]. For those Agile development teams releasing incomplete or buggy software, of which security bugs represent a portion of released defects, those defects will rebound back into the team's backlog and get reprioritized against the current backlog of features in the next release – adding points back in that had been burned down only a few weeks ago. Defects have real Agile risk consequences up to and including sprint bloat risk (too many points in the sprint), decreased feature coding velocity, and the added mental burden of having to conduct regression for fixes against new software in a DEV branch which may not be fully developed at the time. Software quality is not a state; it's a maturation process. By embedding security control overlays into the software requirements, the team takes a step towards maturing DevOps execution, by removing some of the natural volatility found in security requirements *(e.g., the percentage change due to addition, deletion, and modification of a requirement over time)*,[12] and slowly pivots to a culture of communication and collaboration with respect to security requirements early in the design phase. Initiating control conversations in the QA process is too late.

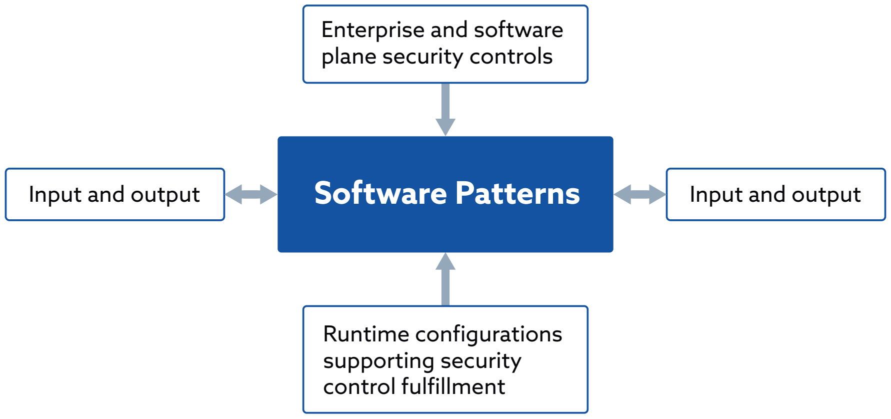
Although this paper's content can aid any number of roles, it is tailored to the security architecture role and an architect's point of view. The MAP paper includes a companion work instruction (Appendix D) to help form control overlays for architectural patterns. The work instruction goal is to create a non-statistical decomposition/re-composition approach to security architecture that

[11] Boehm, B., & Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Computer*, 34(1), 135–137.
[12] V. Suma, B. R. Shubhamangala and L. M. Rao, "Impact analysis of volatility and security on requirement defects during software development process," International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), Chennai, 2012, pp. 1-5, doi: 10.1049/ic.2012.0145.

remains economical and lightweight without sacrificing the basics of control selection in the design process. In the following section, each software architecture pattern found in Section 5, will have a corresponding security control overlay (Example mapping: one to one, many to one, one to many etc.). From a design standpoint, the expanded concept is that of software architecture building blocks (collections of related software patterns) with the control states already known. Logic dictates that architecture building blocks can be assembled into solution building blocks[13]. Through later software requirements decomposition, the technical requirements and technical security configuration reveal themselves simultaneously. A consumer of this working paper uses the software patterns and matching overlays to create a microservice software application and rough in the approximate control state before any agile story writing, story pointing, and software development occurs.

## 6.1 Introduction to Overlays

Security control overlays enforce business and security policies to reduce risk to an acceptable level. Control selection is a balance between business need, time to market, and tolerance for risk. The security overlay wraps a software pattern, and although it might introduce a larger security control coverage scope, only one security overlay applies to one pattern. The additive effect of controls acting at different locations and layers within the microservice architecture yields software defense in depth.


Enterprise and software plane security controls
Input and output
Software Patterns
Input and output
Runtime configurations supporting security control fulfillment

Some overlays create policy enforcement opportunities within a microservices architecture. However, not every pattern and its' associated overlay creates a policy enforcement (PEP). A PEP determines whether or not to permit or deny specific computational or transactional actions. When assembling software patterns into a proposed application and then cataloging the control overlays, assess whether or not other controls within and across overlays can compensate for the failure. Controls in other software patterns overlays might backstop the primary control, reveal themselves to be duplicative, and potentially not necessary. Treat the following overlays as part of a software pattern building block. Use the patterns to assemble the high-level software architecture and then catalog the overall control state to appreciate control coverage or areas absence. The security control overlays apply a specific discrete set of controls to a software pattern as captured in the associated table. The left table column identifies the software pattern and the right table column the pattern's control set.

[13] The Open Group. The Open Group TOGAF Standard version 9.2. The Open Group: Document Number: C182 Found at https://pubs.opengroup.org/architecture/togaf9-doc/arch/index.html. Access on 22 FEB 2021.

### 6.1.1 Service Overlays

Services typically implement a specific functionality and provide the functionality to service consumers through APIs which are accessible using REST/JSON, gRPC/Protobuf  and SOAP/XML protocol and formats. Legacy web services are still providing their functionality via APIs using heavier SOAP/XML protocols. APIs play a very critical role in microservice architecture to build the complex application system and simplify the application integration. Currently most of the data breaches are happening due to lack of API security by design. APIs are a very common target for attackers as APIs expose service functionality along with sensitive data such as Personally Identifiable Information (PII).

Service overlays provide security control guidance to protect the services at service instance, method and protocols. API services should have a mechanism for each request to be authenticated and authorised to access the APIs using protocols such as OpenID Connect, OAuth 2.0 and SAML. In general, end-user context can be verified at the API gateway and allow the access on downstream services using service level trust by using service accounts over mTLS but it can be too permissive and can be mitigated by maintaining the user level security context along with service level trust. Logging and monitoring of API access can be played as a detective control to detect security concerns. Rate limit controls would prevent DDoS attacks on API services. To prevent the threats inside the network,  service to service communications can rely on secured communication such as HTTPS and appropriate authentication/authorization between internal services or internal communication using internal pod localhost IP communication.

Services Overlays are focused only on service level patterns such as Facade, Strangler, Circuit breaker and Adapter. Service Overlays are discussed here to ensure REST, gRPC and SOAP based API services work as intended.

**REST** uses HTTP with TLS to perform operations on service endpoints securely. REST uses JSON standard format for consuming API payloads. Input validation should be done for length, range, format and type. Content type should be verified for request and responses. Input request size should be restricted to fixed size and requests should be rejected with 413 error when request payload exceeds the size. Logging and monitoring should be enabled for all failures of input validation and token validation. Sensitive data should be transferred via request body or headers and not in URL. General origin setting should be avoided in CORS headers.

**gRPC** is a high-performance and light weight RPC framework. gRPC uses HTTP/2 to perform operations among services. gRPC uses protocol buffers for message interchanging format. Due to the binary format of messages in gRPC, payload inspection and content validation should be done thoroughly to separate out malicious payloads and avoid accidental disclosure of sensitive data. gRPC supports multiple languages so selection of better memory management language plays a key role to avoid memory management vulnerabilities. Developers should avoid usage of default insecure communication options like “InsecureChannelCredentials” in their code.

**SOAP** is a XML based messaging protocol to exchange information among services. SOAP built-in standards like Web Services Security use XML authentication, XML signature and SAML tokens to secure messaging. SOAP/XML payloads should be validated against associated XML schema definitions. It should define a maximum character length limit for every input parameter and should define strong strict validation patterns for fixed parameter patterns.

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| Facade | Authentication mechanisms should be implemented in various internal subsystems. | IA | IA-3(1), IA-5(2), IA-9(1) |
|  | Object level authorization should be implemented in various internal subsystems that access a data source using the user's input | AC | AC-3(1,7), AC-6(1) |
|  | Internal subsystems should be secured by firewall when all requests coming through Facade and Facade IPs are only allowed by firewall | SC | SC-7(11, 15) |
|  | Ensure the availability of internal subsystems | SC | SC-6 |
|  | Avoid excessive data exposure from internal subsystems | AC | AC-4 (5,6,19, 22) |
|  | Failure of internal subsystems management like ineffective auditing, multiples owners, legacy code and over simplification | PL | PL-2(3) |
|  | Secure every permutation of service or function in interaction flow | PL | PL-2(2) |
|  | Input validation length, format, type, Define and allow only appropriate request size. CORS, Security headers | AC | AC-4 (12,14,19) |
|  | Avoid binding client provided data to data models without proper properties filtering based on an allowlist | AC | AC-4 (15,19,20) |
|  | Use client side TLS 1.2+ which will ensures authenticate the server and encrypt the data | SC | SC-8(1,3) |
|  | Use **gRPC** built-in authentication mechanism like SSL/TLS, ALTS and Google token based authentication | IA | IA-3(1), IA-5(2) |

| | | | |
|---|---|---|---|
| | Enforce use of **gRPC** credentials objects which can be used to create the entire channel(Channel credentials) or individual api call(call credentials) | IA | IA-9(1) |
| | **gRPC** supports multiple languages and try to use memory-safe languages to avoid high impact memory management vulnerabilities such as remote code execution due to buffer overflows | AT | AT-2(1) |
| | **SOAP/XML** payload parsing and schema validation at Facade interface to avoid common vulnerabilities( SQL injection and Cross Site Scripting), DDTs and XML schema vulnerabilities. Validation against malformed XML entities, XML bomb attacks | AC | AC-4(5,6,12, 15,19,20) |
| Strangler | Object level authorization should be implemented in various legacy subsystems and new micro services that access a data source using the user's input | AC | AC-3(1), AC-3(7), AC-6(1) |
| | Authentication mechanisms should be implemented in various internal subsystems | IA | IA-3(1), IA-5(2), IA-9(1) |
| | Enforce the use of HTTPS endpoints on legacy applications; to secure the transmitted data, allowing clients to authenticate to the server and preserve the integrity of the transmitted data | SC | SC-8(1,3) |
| | Implement and enforce the use of API keys to avoid the impact of DDoS attacks | SC | SC-5(1) |
| | Failure of legacy applications and new application management like ineffective auditing, multiples owners, legacy code and over simplification | PL | PL-2(3) |
| | **SOAP/XML** payload parsing and schema validation at legacy backend services | SC | AC-4(5,6, 12, 15,19,20) |
| Circuit Breaker | Risk to system cascade failure due to misconfiguration of policies | CM | CM-2(1), CM-3(2,3), CM-6(1) |
| | Risk to failure of Monitoring of downstream services | CA | CA-7(3) |

| | | | |
|---|---|---|---|
| Adapter | Input validation length, format, type, Define and allow only appropriate request size | AC | AC-4(12, 14,19) |
| | Authentication mechanisms should be implemented in various backend services | IA | IA-3(1), IA-5(2), IA-9(1) |
| | Avoid binding client provided data to data models without proper properties filtering based on an allowlist | AC | AC-4 (15,19,20) |
| | Ensure the availability of internal subsystems | SC | SC-6 |
| | **SOAP/XML** payload parsing and schema validation at legacy backend services | SC | AC-4(5,6,12, 15,19,20) |

Service overlays also provide security control guidance to apply in the software development process for building, testing and deploying secure API services irrespective of patterns.

Coding standards should be defined and applied (automatically as much as possible) in the build phase of the software development process.

- Choose the right bootstrap frameworks
- Ensure code quality according to assigned metrics
- Maintain full unit test coverage
- Defect tracking according to assigned metrics
- Choose a singular code base with multiple deployments vs multiple code bases
- Utilize Service Repositories (e.g. Swagger definition)

Continuous Improvement and Continuous Delivery (CI/CD) pipelines ensure that run-times in lower environments are the same as those found in production.

- Security testing baseline: vulnerability scanning, tracking and remediation
- In the IDE (shifted-left static testing performed by the DEV as a precondition to branch merge)
- Across the repository (static enterprise plane testing for software vulnerabilities)
- In the QA environment (dynamic enterprise plane testing of a functioning microservice application and its containers)
- In production: assessing container security posture runtime vs base

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| All | Coding standards should be enforced and tracked as per assigned metrics | SA | SA-15 (1,2,4,7) |
| | All DEV and QA environments should be similar to production environments | SA | SA-3 |
| | Ensure that all the security assessment activities (build and run-time) of microservice are performed and automated | SA | SA-11 (1,2,5,8) |
| | Ensure that vulnerability scanning should be performed in software development cycle | RA | RA-5(3,9,10) |

### 6.1.2 IAM Overlays

IAM (Identity and Access Management) maintains the identity of the user and servers after a first validation to avoid derivative attacks due to lack of correct handling of the duration of sessions such as Clickjacking and Cross Site Forgery, identify the user, and allow / register the activity and block for a period of time and in case they are really an attacker, permanently disable.

Within the code libraries that are used to establish IAM functionality we have Auth, OAuth & OpenID. One can enable One Time Password and multi-factor authentication functionality by adding part of the user's information to the session and then sending this key as part of the authentication in ticket handling systems such as Kerberos or session tokens such as JWT.

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| Offload | SAML/Oauth token update/refresh mechanism This control allows to centralize user access with a user-friendly interface and keep user information throughout the session. | SC | SC-23(3) |
| | Container fleet with adjacent security logging interconnections this work also like 2FA avoid attack like cross site forgery request petition | AU | AU-15 |

| | | | |
|---|---|---|---|
| Route (Routing) | IAM policy defines routing like some routing and switch networking providers. Load balancing routes avoid attacks like DDOS or DOS in level 3 of the OSI model but with IAM you can apply at layer 7 | AC | AC-4 |
| | Prevalidation references of the API try to consume other APIs using headers like CORS origin & Content Security Police, avoid sending data through an invalid route (the APIs and the role of the user to try to consume an API is not allowed for that) to save resources. | AC | AC-4(30, 31) |
| Aggregation | Group APIs using Orchestration and access them using IAM. This authentication system allows delegating the load of resources of administering the session after the Docker container has been deployed, avoid overflow in Docker containers and improve the high viability of the APIs | AC | AC-29 |
| Cache | IAM credentials kep in Docker-credential-secretservice for a time lapse like a cache, avoid the send continue requests to validate the users to the IAM | IA | IA-4(9) |
| AuthN | Use self signing client digital certificates for offline services and trigger using RPA, help to validate logins and captcha without the interaction of users that consume of APIs, like 3 Factor authentication pieces of "salt" (piece unique identicator of the user) avoid falsification of the user and machines. | AC<br>IA<br>SC | AC-4(17)<br>AC-4(21)<br>AC-4(3)<br>IA-5(2)<br>SC-23(5) |
| AuthZ | Add IAM user information to access tokens like JWT or Kerberos which help validate the user in functions of the class and prevent successful attacks like cross site forgery. | SC | SC-23(3) |
| | Implement XACML, RBAC, ABAC for various subsystems. Allow subdivision of authorization control by subgroups instead of a one formal federation system to help in security redundancy. | AC<br>IA | AC-10<br>IA-5(8) |

| | | | |
|---|---|---|---|
| Facade | Enable messages when IAM fails, overload and successfully send messages in the front end and in HTTP/HTTPS/TLS.<br>Inform the user as to the actual state of the session to avoid confusion and multiple failure attempts or incomplete requests. | IA | IA-4(4)<br>IA-4(6)<br>IA-3(1) |
| Circuit Breaker | Integrate DDOS protection code libraries with IAM and establish a message chain between them to keep track of any event raised as a possible attack. | SA | SA-4(2)<br>SA-8(8) |

### 6.1.3 Network Overlays

Network functions are typically implemented in gateways which implement offload, aggregate, routing, proxy and cache patterns. Network overlays ensure information and information systems are protected by these patterns. From an operations centric view, they can control access by source IP addresses (Who), destination URLs (What), and communication protocol and encryption (How). From a data centric view, they can control whether data can be accessed with or without encryption (How). From a people centric view, they can control access by blocking source IP addresses outside of a company's intranet or source IP addresses allocated to a country under commerce embargo.

Increasingly, microservices are aggregated to offload many essential services e.g. IAM services of individual services and dependent on service routing (typically using ISTIO) to reach the individual services. Aggregation is done by sharing the URL endpoint among the services. Routing is done by the different URI paths of the target services. The result is often a mesh or a network in the application layer. The design of the mesh requires careful security consideration to avoid adverse consequences. For example, a typical mistake is routing internal service requests to an external endpoint. The general principles in mesh design are:

- Endpoints for data of different security classification must be segregated
- Endpoints for users of different security classification must be segregated
- Services serving different trust zones (e.g. internal vs. external) must deploy sufficient boundary controls.

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| Offload | The pattern takes up the standard layer-4 communication protection (TLS) for all microservices. It provides confidentiality and integrity with cryptographic function; deny incoming traffic which has no adequate cryptographic protection. Without adequate cryptographic protection, data confidentiality and integrity are at risk. | SC, AC | SC-7(11), SC-8(1), SC-12(2,3), AC-17(2) |
|  | Typically, the pattern also provides authentication to one or both communication parties. This lower-communication-layer authentication is no replacement for application-layer authentication. | IA | IA-3(1), IA-5(2) |
| Proxy | Instead of exposing all microservices in one trust zone to another, the pattern is used to expose only selected zones. It bridges communication across trust boundaries. | SC | SC-7(5,11,17) |
|  | Restrict communication on specific service flow. Excessive data exposure may result without these controls. | AC | AC-4(2,6) |
| Aggregate | The pattern aggregates different microservices into one. It has the risk of excessive data exposure. Protection adequate for one service may not be for another. Therefore, services for data of different security classifications must not be aggregated | AC | AC-4 (2,6,8, 11,12,14) |
|  | To maintain function level authorization, services for users of different security classification must not be aggregated | AC | AC-3 (3,4,13, 14,15) |
|  | Fine-grain authorization policies must be defined and enforced. Aggregating services must not break object-level authorization | AC | AC-6 (1,2,3, 4,9,10) |
|  | Services must be periodically assessed on the impact of data exposure. Change in data security categorization or classification must be reflected in change of service aggregation. | RA | RA-2(1) |

| | | | |
|---|---|---|---|
| | Services must be periodically assessed on the user classification. Change in user classification must be reflected in change of service aggregation. | RA | RA-5(5) |
| Route | The pattern channels different microservices into one before routing. It has the risk of exposing one service for another and creating excessive data exposure. Services on data of different security classifications must be routed differently | AC | AC-4 (2,6,8, 11,12,14) |
| | To maintain function level authorization, services for users of different security classification must be routed differently | AC | AC-3 (3,4,13, 14,15) |
| | Fine-grain authorization policies must be defined and enforced. Channeling multiple services must not break object-level authorization | AC | AC-6 (1,2,3, 4,9,10) |
| | Services must be periodically assessed on the impact of data exposure. Change in data security categorization or classification must be reflected in change of service aggregation. | RA | RA-2(1) |
| | Services must be periodically assessed on the user classification. Change in user classification must be reflected in change of service aggregation. | RA | RA-5(5) |
| Cache | The pattern caches data for different microservices. It has the risk of exposing data of different sensitivity and creating excessive data exposure. Data of different security classifications must not be served by the same cache pattern. | AC | AC-6 (1,2,3,4, 9,10,12) |
| | The pattern alleviates high data availability requirements from microservices and defends them from data delivery resource limitation. | AC | AC-21(2) |

In addition to the specific security functions by the individual patterns, the network patterns are at advantage implementing network-based test and monitoring such as distributed API tracing. Network-based test and monitoring can help:

- Discover and map API relationships between resources
- Generate API topology maps based on alerts to streamline troubleshooting
- Mature into visualization and navigate resources based on topology relationships
- Find failure points and areas of service disruption vulnerability.

### 6.1.4 Monitoring Overlays

The monitoring control overlay is intended to provide event logs, metrics, and alerts to ensure the security and availability of the microservice(s), container(s), and the underlying platform or runtime environment.

The control overlay is focused on applications and microservices, as well as the containers and underlying platform. With that intent in mind, consider the following scenarios:

- When a microservice generates a security event, that event needs to be captured and logged.  Optionally, an alert may be required.
- When a container is instantiated, that event may need to be captured and logged.
- When a microservice performance metric exceeds a predefined threshold, the underlying container or platform may need to scale-up/scale-out to facilitate the increased load on the microservice.
- Operational teams need to be informed when the solution ecosystem is running in an inefficient state.
- Business stakeholders may want to be informed when scale-up/scale-out conditions are invoked, particularly when cost becomes a factor.
- Incident response teams may require access to verbose log data for investigative purposes.

Although the above is not meant to be an exhaustive list of scenarios, it hopefully provides some context for the controls required for this particular overlay. The primary control function may reside at a control or enterprise plane level. However, there are often specific configurations required at the microservice level to enable the control. For example, if the container platform or runtime environment has a verbose logging facility, but the microservice is not configured to log events (or the log level is improperly set), then the control fails. It is assumed that developers of microservices should be focused on business functionality, and logging of events is simplified to writing those events to an output stream. That event stream is then captured, collated or aggregated, and forwarded or routed to platform logging facilities, syslog sinks, dashboards and other analytics and monitoring tools.

As we explore the audit, monitoring, and alerting controls, we consider the following guiding principles when selecting our set of controls:

- Security and key operational events must be logged
- Logs must be trustworthy (e.g., immutable, cannot be repudiated) to support forensics and root cause analysis
- Logging mechanisms must be resilient from failure
- Application developers must not be concerned with how the logs are captured or routed

When designing and implementing a monitoring control, we can leverage NIST SP 800-53 for further guidance on the expectations for that control. The following table helps to identify the proposed controls for the application or microservice monitoring overlay:

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| All | Logging supports forensics, investigations, performance, operational resiliency, and root cause analysis.  Favor quality of information over quantity of telemetry data. | AU | AU-2<br>AU-3<br>AU-12 |
| | Monitoring changes to application and container configuration is critical to forensics and root cause analysis of outages. Alerts are generated when key component configuration element changes affect the security posture of the solution. | CM | CM-2<br>CM-3(5) |

The following table helps to identify the controls required for the container, platform, or enterprise plane monitoring overlay:

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| All | Logging supports forensics, investigations, performance, operational resiliency, and root cause analysis. Favor quality of information over quantity of telemetry data. | AU | AU-2<br>AU-3<br>AU-9<br>AU-12<br>AU-16 |
| | Monitoring changes to proxy configuration is critical to forensics and root cause analysis of outages. Alerts are generated when key proxy configuration elements affect the security posture of the solution. | CM | CM-2<br>CM-3<br>CM-12 |
| | Continuously review, analyze, and correlate data into actionable information. Respond to critical events. Present findings, actions, and trends to stakeholders via alerts, reports, and dashboards. | CA | CA-7 |
| | Once critical security events are identified, there must be dedication from the organization to understand what happened, why it happened, and how to prevent the critical event from happening again. | IR | IR-4<br>IR-5 |

These individual controls do not work in isolation, but collectively address the perspective or view of the architect to ensure that appropriate monitoring has been established.

As an example, retaining an immutable log of an event requires having a baseline expectation for the monitor (CM-2 Baseline Configuration), ensure that the baseline functionality remains viable and has not been compromised (CM-3(5) Configuration Change Control), and that the telemetry and log data is sufficiently protected from corruption (AU-9 Protection of Audit Information.)

The monitoring control overlay applies to all of the patterns from section/chapter 5.

Finally, the application development and operations teams must be aware that monitoring controls themselves may require a monitor or alert mechanism. This is an opportunity for Product Owners and Managers to review and analyze information, such as Key Performance Indicators (KPIs), security metrics, and trends across those metrics, to ensure that the business operational needs are being met, there is sufficient separation of duty between contributors, and that there is a prioritization of issues which are impacting functionality and security posture.

### 6.1.5 Cryptologic Overlays

Within Microservices, encryption is used to protect data. The type of encryption depends on the data, environment and any threats. Detailed threat modelling can be conducted to identify attack scenarios and potential attack vectors, sensitivity of the data, applicable regulatory compliance or legal requirements and the impact to company reputation due to a potential data breach. With Microservices, we see three states of the data to be protected that apply to a variety of microservice patterns such as  - Offload, Routing, Aggregation, Cache, AuthN, AuthZ, Facade and Adapter patterns.

- In Transit (Network)
- In Use/Processing (CPU, Memory, Cache)
- At Rest (Storage)

#### In Transit

End users authenticate through an external facing endpoint.  The various communication paths such as access from external services to microservices, users to microservices as well as between the microservices itself also needs to be protected. With so much communication between microservices, it is important to enforce data protection in transit via mutual authentication over Transport Layer Security (mTLS). HTTPS (TLS) is designed to ensure privacy and data integrity, through an X.509 certificate which serves two functions:

- To grant permissions to use encrypted communication via Public Key Infrastructure (PKI)
- To authenticate the identity of the certificate's holder.

Microservices communicate to authorization servers and other microservices. It is likely that secrets are used during communications. These secrets might be an API key, or a client secret, or credentials for basic authentication. The secrets should be protected by encryption and should not be checked into the source control system.

### In Use/ Processing/ In Cache

The protection of data in use/cache is more complex when compared to data at rest and data in motion because "data in use" must remain accessible when needed. Protection of data in cache involves - confidentiality of data in the cache and the confidentiality of data while in motion between the cache and the application while it is using the cache. Data in the cache could be partitioned or segregated and each partition should be protected via strong authentication and tight (& granular) access controls.  Each partition could be further encrypted via encryption keys that are available to only authorized entities. The cache data while in motion between the cache and the application can be protected by secure protocols such as mTLS, TCP, HTTPs. The organizations also should be able to log, track, report, monitor and alert to detect and prevent potential threats. A cache can be further protected by:

- Specifying the limit on the maximum amount of memory that can be used;
- Configuring keys in a cache to have an expiration time, after which it is automatically removed from the cache; and,
- Cache invalidation (ensuring stale data is removed, automatically evicting key-values based on policies)

### At Rest

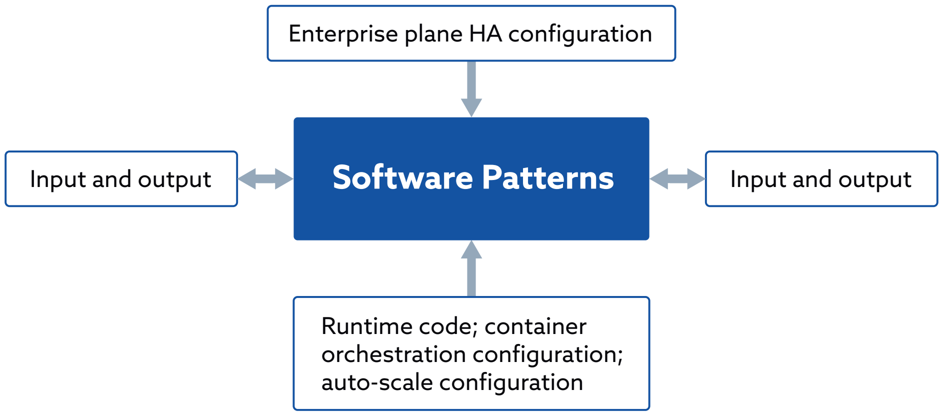
Once the access is authenticated, it is important to establish an authenticated user's security context before allowing access to resources/ data provided by a service via an access token. The access tokens must be digitally signed to prove authenticity of the issuer and the integrity of the token. The token can also be encrypted to provide confidentiality of data (such as for sensitive financial data). Tokenization substitutes sensitive data elements with a non-sensitive equivalent referred to as the token.

| Microservice Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| Offload | The pattern provides Transport layer protection (mTLS), confidentiality and integrity using cryptographic function; Security functions such as token validation, encryption, SSL certificate management need proper management. A certificate needs to be properly configured and deployed and  be refreshed  periodically before it expires. It needs to be tested before being deployed. | AC | AC-17 |
| | | IA | IA-3 (1) |
| | | SC | SC-23(3) |
| | | CM | CM-3(6) |

| | | | |
|---|---|---|---|
| Route (Routing) | Routing pattern provides a layer on top of the network that means your application code can connect to services rather than individual containers<br>The pattern leverages Cryptography, Encryption, Coarse/Fine grained access control, Token entitlement payload | AC | AC-10 |
| | | SC | SC-23(3) |
| Aggregation | The Aggregator pattern receives a request via service, subsequently makes requests of multiple services, combines the results and responds to the initiating request. The Aggregator Microservice can collect data from multiple Microservices and return the results to the consumer.<br><br>The pattern uses security controls such as Cryptography, Encryption, Coarse/Fine grained access control, Token entitlement payload for secure communication | AC | AC-10 |
| | | SC | SC-23(3) |
| Cache | Caching patterns increase availability, improve performance, reduce backend load, or decrease downtime by storing data for re-access in a device's memory.<br><br>The caching pattern uses security controls such as Cryptography, Encryption, Coarse/Fine grained access control, Key-value pair integrity verification at runtime (JSON web token) | AC | AC-10 |
| | | SC | SC-23(3) |
| AuthN | AuthN patterns provide the capability to validate the identity of the entity.<br>The AuthN pattern uses key signage (Digital Signature), JSON web token, encryption policies for securing the data for securing this pattern. | IA | IA-9 |
| | | AU | AU-10 |
| | | SC | SC-23(3) |
| AuthZ | AuthZ patterns provides the capability for verifying if the entity is authorized to access specific information or is allowed to execute certain actions.<br><br>The AuthZ pattern uses key signage (Digital Signature), JSON access token, encryption policies for securing the data for securing this pattern. | SC | SC-23(3) |

| | | | |
|---|---|---|---|
| Façade | Facade microservice pattern hides the complexity which is required to make a request to an external service or provide a result-set based on some business logic. The pattern uses security controls such as Cryptography, Encryption, Coarse/Fine grained access control, Token entitlement payload | AC | AC-10 |
| | | SC | SC-23(3) |
| Adapter (wrapper/ translate/ transform) | Adapter pattern provides a way to identify how to realize relationships between classes and objects in a simple way. This pattern is used when an incompatible module needs to be integrated with an existing module without making any source code modification. The pattern uses security controls such as Cryptography, Encryption, Coarse/Fine grained access control, Token entitlement payload | AC | AC-10 |
| | | SC | SC-23(3) |

## 6.1.6 Microservice Resiliency and Availability Overlay


Enterprise plane HA configuration
Input and output
Software Patterns
Input and output
Runtime code; container orchestration configuration; auto-scale configuration

### Microservice Resiliency and Availability

The scope of the Microservices Architecture Patterns paper chose to thread attestation to the importance of governance risk and compliance (GRC) through the paper's sections rather than elaborate GRC topics as a stand-alone complex. Elements of GRC can be found throughout the security overlay sections. Remaining consistent with shifting left towards the software and away from the program management aspects of GRC, microservice resiliency and availability is an amalgam of effects generated by inherited architecture attributes from the enterprise plane, such as backup, recovery, high-availability, and horizontal scale plus software resiliency design tactics further bolstering performance under load. Resilience is often spoken of as a singular capability when it represents a meta-capability composed of three specific architectural principles.

- *Absorptive capacity* – can absorb shock or pressure above what is considered normal application activity
- *Adaptive capacity* – When under load, the application can deform in a predictable way to allow continued operation by offering a lower level of service guarantee
- *Restorative capacity* – Following episodes of pressure and adaptation to application load in excess of what monitoring reveals to be normal, the application can self-restore to its prior state.

Availability is the measured period of time that an application functions in a multi-user multi-mode state and is able to perform its agreed function for a specified period of time. An application can be "up" or "on" but not available, such as during periods of planned maintenance, known as planned downtime. Availability[14] equations factor in planned downtime, and represent availability as a percentage of time (ITIL Equation[15]). In a microservice context, availability is the summation of a chain of microservices arranged to deliver a specific business capability. In a simple example,

(Service 1 (.99 available) x Service 2 (.99 available) x Service 3 (.99 available) =.97)

When assessed individually all three microservices are 99% available. As transaction chains of microservices lengthen, end-to-end transaction availability diminishes. Employ platform resiliency tactics to accommodate chaining effects. When designing microservice applications, conserve the number of interface hops in a transaction chain. Utilize enterprise plane platform capability to offset expected declines the calculated service availability of transactions traversing microservice applications. Pursue high availability platform capability such as redundancy (N+1) or higher as business requirements dictate to prevent cascading microservice failures. Use load balancing at the enterprise plane, and at the head-end of the software plane, and continue to rely upon traditional backup and recovery capability – with caveats. Container platforms follow traditional tactics, but providing for microservice backup and recovery has more in common with high volume data backup and recovery than the platform microservices host upon.

## Microservice Software Resiliency and Availability

The Twelve Factor Application framework created by developers at Heroku in 2012[16] proves itself over and over as the microservice architectural style matures. A core tenet of the microservice architectural style is that application services have a single responsibility. Build and deploy one business capability per container operating system in order to assemble those things that change under similar circumstances, to maintain separation from other services that change for unrelated purposes. Loose coupling and independence from other application components ensures that communication occurs only through an API to solve more complicated business operations. For example, consider a business process workflow that requires an order update capability. A single microservice hosted in a single docker container, coded to perform a record update action represents

[14] Information Technology Infrastructure Library (ITIL). IT Service Management and the IT Infrastructure Library (ITIL). IT Infrastructure Library (ITIL) at the University of Utah. Retrieved June 15, 2021, from https://itil.it.utah.edu/index.html.
[15] Rance, S. (2017, June 22). *How to Define, Measure, and Report IT Service Availability.* ITSM Tools. https://itsm.tools/how-to-define-measure-and-report-service-availability/.
[16] Wiggins, A. (2017). *The Twelve-Factor App.* https://12factor.net/.

one business capability (i.e. update order) manifested by a single microservice called to update an order record with new data. Under the conditions of this level of service atomicity, it's clear that microservice applications represent a constellation of microservices communicating among each other to perform more complicated business tasks. Within the microservice architectural style, isolation of business processes into singular microservices is a principle means to scale out processing to accommodate increasing load — it is a key driver of technical resiliency via inheritable container and clustering platform availability. Container orchestration for deployment at scale permits maintenance, security, and governance of master container images and file definitions in an image repository. The orchestration function manages container load, offering the ability to size memory and CPU according to application performance not container performance. Last, orchestration offers the ability to auto-scale (horizontal scale) new container instances as according to application load thresholds.

Within the software development design process, principal developers and software architects can engineer software resiliency capability when the platform capability cannot meet business requirements alone. Specific tactics include aggressive or opportunistic TCP timeout and connection resetting, fall-back coding to provide service in lesser functional state, application of circuit breaker or wrapper software architecture patterns to shape and control surges in traffic, and code specifically for thread isolation and/or multi-threading. The emphasis needs to be on stateless transactions that will be eventually consistent. Additional platform tactics to increase application and data availability include software load balancing among multiple instances of the same microservice, apply local and remote caching software patterns usage for data redundancy, use queuing of data, and if necessary refactor a facade or cache software pattern into a bulkhead to isolate services into a data compartments to break cascading failure potentials among critical microservices. Contingent upon business process resiliency and availability requirements, a number of tactics are brought to bear to establish environmental variables, configuration, and backing services independent of the actual microservice software coding. Capability in the enterprise plane and the software plane drive microservice resiliency and availability.

## Microservice Backup and Recovery Tactics and Trade-offs

Microservice applications cannot be recovered in a consistent state from individual and independent backups[17]. Microservice application state changes place backups at risk. Connections between different entities managed by different microservices need to remain valid and intact. Microservice backup and recovery is tightly coupled to data repository transaction restoration. Even snapshot backup and transaction logging cannot restore current state by the time of the snapshot, the microservice data currency has already moved forward into the future. Even under the best high-availability transaction logging relation database system a 15 minute lag remains common. In order to ensure data consistency, data record locking and a simultaneous snapshot of microservice state occurs during the backup period. Locking the application and data constraints read-write service availability permitting only read operations. Depending on the duration of the backup (which is bound by the least performant data store and enterprise plane network design), clients whose requests to change the state will be denied or delayed until the backup of the entire application is completed.

[17] Pardon, G., Pautasso, C., & Zimmerman, O. (2019). *Consistent Disaster Recovery for Microservices: The BAC Theorem.* IEEE Cloud Computing. https://design.inf.usi.ch/sites/default/files/biblio/bac-theorem.pdf.

## Eventual Inconsistency with Full Availability versus Consistent Backups with Limited Availability

When making an independent backup of the state of each microservice service, it is possible that one service will be backed up before the event is added to the backup transaction log, while the other service will be backed up after the corresponding event is added to its backup transaction log. When restoring the services from backup, only one of the two services will recover its complete log. The global state of the microservice architecture will thus become inconsistent after restoring it from backup. Consider how a broken link manifests in Internet hypermedia following backup and restoration. A broken link is a relative reference that can no longer be followed from page A to page B, or target A to target B. The resultant broken URI happens when there is no related reference. In the microservice context, microservice B remains consistent with A until it becomes unavailable. After recovery from an obsolete backup, B does not have the state that corresponds to the latest events logged by A breaking the transaction chain. If the business requirements dictate that while the backup is running, no events can be added to the microservice logs, this requirement effectively reduces the availability of the microservices for clients that need to perform some state transitions/ write operations. The trade-off avoids inconsistency by coordinating the backup of microservices services to ensure that a snapshot is taken either before the interaction takes place or after the effects of the interaction have been logged on both sides. However, the trade-off violates the independence of the two microservice as it introduces tight coupling into their operational life cycles. It is for this reason microservice application backup and recovery should occur at the pod (many related containers), or cluster (many related pods) and not for individual microservices. Preferentially, microservice applications deploy as a load-balanced "A" side and a mirror-image "B" side where one can be kept in read-write while the other side is in read-only for maintenance, patching, or restoration (and vice-versa). Backing data services will need to utilize caching and queuing to mitigate the eventual inconsistency. RESTful services are always data inconsistent and that is a trade-off of choosing REST method transaction communications over a protocol-based data transport. Microservice state magnifies data inconsistency. There can not be autonomous persistence (or polyglot persistence, for that matter) and at the same time consistency after recovery from a backup. On the other hand, "eventual consistency" as it is implemented today does not work well. If two systems are independent and counting upon changes to ripple through, then the possibility exists where microservice backups do not change at the same rate. Microservices backups must occur at the same time in order to be consistency, which in practice means one shared data store for all of them, or consistent backups across separate data stores[18].

Production performance analysis requirements dictate that all broken and orphaned microservice interface connections should be logged as they are detected in order for operators to decide which tactic to follow when attempting to restore the consistency of the application during a period of application discontinuity, or after a disaster declaration. Common DevSecOps tactics include;

[18] Github. (2019.) *Guide on Microservices: Backups and Consistency | Consistent Disaster Recovery for Microservices - the BAC Theorem.* dhana-git/Guide on Microservices: Backups and Consistency.md. Retrieved August 11, 2021, from https://gist.github.com/dhana-git/3dda5326b3bd15a93d3389a6c30d3000.

*Do nothing*

The system and its users can accept that some parts of the system can be inconsistent.

*Rely on cached data*

The system relies upon multiple layered caches for data. If one cache hit fails, the system can resort to a secondary cache, or make a call to the source of record. Otherwise, the system could process the events with read-only warnings to the end user.

*Human intervention – orphaned process state*

An orphan processes can cause validation errors and, consequently, the rejection of the incoming transaction events and messages. A performance analysis requirement is to regularly run a validation service and/or database consistency checker.

*Human intervention – missing process state*

For those events that are easy to reproduce from the source, replay missed events. For those events not easily reproduced from the source: manual intervention might be needed to (re)apply the missing state in the form of one or more user commands or accept that events are lost.

## Service Continuity Considerations

Consider investing in separate enterprise plane high-bandwidth networks for backup and recovery to ensure that backup and restoration traffic does not have to compete with day-to-day organizational traffic. *High-availability (HA) data layer:* With an HA backup network in place, the backup process happens continuously, and the recovery can theoretically be instantaneous. However, a high-availability data layer for every microservice would be an expensive solution which would increase the operational complexity of the system and would not be convenient for many aspects of an organization. Thus, there would be the need to compare the cost of dealing with the eventual inconsistency of the application after its recovery. As stated previously, backup at the pod and cluster level. Second, consider applying distributed snapshots (or "snaps"). Snaps are necessary for global state detection and check-pointing of the entire microservice architecture. It is possible to coordinate the various microservices as they store their state into a snapshot for backup purposes. When using snapshot data storage tactics, each local database contains the current state of its microservice. However, this limits the flexibility of independent microservice deployment, operation, and evolution[19]. During the software design phase, direct programming to recover the state of a component, starting from a known initial state, and by replaying every message it received in the same order they reached the component/entity before the period of unavailability. Apply caching patterns, cached replicas, and reliance on re-reading in environmental and configuration data from files or repositories rather than relying upon the current runtime state.

[19] Pardon, G., Pautasso, C., & Zimmerman, O. (2019). *Consistent Disaster Recovery for Microservices: The BAC Theorem.* IEEE Cloud Computing. https://design.inf.usi.ch/sites/default/files/biblio/bac-theorem.pdf.

| Patterns | Security Functions | Control Families | Primary Controls |
|---|---|---|---|
| Facade, Strangler, Circuit Breaker, Cache, Wrapper | Automate the verification and retention of past and current baseline configurations, configuration settings, environmental settings, for inventory, retention, deviation detection, and accuracy such that system can restore to an operator-chosen state | CM | CM-2(2), CM-2(3); CM-6(1); CM-8(3); CM-8(6) |
| Enterprise Plane | Pursue inheritable high-availability (n+1) redundancy to ensure that adequate capacity is on hand, or available for invocation should the need arise; store and maintain critical image, file definition, environmental and configuration information, separately from runtime assets. | CP | CP-2(2); CP-9(3); CP-9(6) |
| Enterprise Plane | Provide dedicated networks, sub-networks, DNS, and bandwidth placed backup and restoration traffic out of band from business-as-usual network traffic activity | SC | SC-37(1) |

# Summary/ Conclusion

With the advent of cloud computing, DevOps culture, and major vendors promoting their own software construction patterns and representations, it makes for a crowded field with many powerful voices. Vendors both impart the nuances of their platforms and keep their customers engaged using their own language and concepts. Software pattern use has a long and distinguished history, and what is heard is not necessarily net-new but tried and true concepts applied with new methods in new mediums.

"Shift-left" is traditionally a software development concept where testing and tooling is moved closer to the software engineer such that functionality can be tested earlier in the software construction process. The DevSecOps movement created the opening to explore applying security control overlays to traditional software design patterns. Commonly, control discussions tie back to a work plan or program level assessment effort. DevSecOps motivated organizations to recruit, train, and create defensive software development continuing education. Direct introduction of security design concepts into software design tended to ride upon the presence of vanguard principal software engineers and security architects who tackle hard problems in software construction while mentoring software development teams. Within the field of information security, control frameworks mature and evolve outside of the software spectrum and currently attempt to use automation and "as code" capability to drive control policy into the software deployment process, but not into the software design process.

The Microservices Architectural Pattern (MAP) workgroup attempted to span the last leg of "shift-left" by applying control overlays to software before the code exits the development environment. The aspirational goal was to create a framework for thinking about control atomically at the software pattern level so that teams can debate security design while debating software design. In a perfect universe, the MAP software patterns and overlays, when joined as software architecture building blocks, reveal defense in depth as applied to the microservices architectural style. The ideal outcome is Agile stories for security control capability in the same sprint plan as the Agile stories defining software requirements. Instead of relying on higher level pipeline testing, or traditional "shift-left" testing in the software developer's IDE, software patterns plus overlays jump ahead of writing code to the design phase and inform needed requirements and test coverage for new software development.

Introducing security controls early in the design process is not easy. Some security overlays exist in waterfall organizational processes and are fully reliant on mandated enterprise plane platforms. Overlays reliant on policy enforcement point capability for proper control functionality cannot be tested until later application testing phases. Teams may have an 11-day sprint cadence but often deal with multi-week service level agreements for security capability delivery. Despite the obvious constraints, software patterns with security overlays opens an avenue to conjoin security architecture with software architecture. The MAP patterns plus overlay framework is by no means complete, free from defects, and shielded from persistent challenges. Like anything new and novel – iteration is a necessity born from use.

# Appendix A: Acronyms

| | |
|---|---|
| **ACL** | - Access Control List |
| **API** | - Application Programming Interface |
| **GRC** | - Governance, Risk and Compliance |
| **JSON** | - JavaScript Object Notation |
| **JWT** | - JavaScript Web Token |
| **LAN** | - Local Area Network |
| **MAP** | - Microservices Architectural Pattern |
| **SDLC** | - Software Defined Life Cycle |

# Appendix B: Glossary

**Architect**

The individual or organization responsible for the set of processes to deploy and manage IT services. They ensure the smooth functioning of the infrastructure and operational environments that support application deployment to internal and external customers, including the network infrastructure, server and device management, computer operations, IT infrastructure library (ITIL) management, and help desk services[20].

**Architecture**

Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution[21].

[20] CSA. *Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems* (Cloud Security Alliance: 2019) 42

[21] ISO/IEC/IEEE 42010. (2011). *Systems and Software Engineering — Architecture: A Conceptual Model of Architecture Description.* Retrieved August 11, 2021, from http://www.iso-architecture.org/ieee-1471/cm/.

**Architecture Description**

A conceptual model, an architecture description:

- expresses an architecture
- identifies a system of interest
- identifies one or more stakeholders
- identifies one or more concerns (about the system of interest)
- includes one or more architecture viewpoints and one or more architecture views
- may include correspondences
- may include correspondence rules
- includes one or more architecture rationales[22]

**Architectural Pattern**

A general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design patterns but have a broader scope. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and minimization of a business risk.[23]

**Availability**

The ability of a configuration item or IT Service to perform its agreed Function when required. Availability is usually calculated as a percentage. This calculation is often based on Agreed Service Time and Downtime. It is best practice to calculate availability using measurements of the Business output of the IT Service.[24]

**Business Owner**

A Product Ownership role that represents the person who is accountable to the Business for maximizing the overall value of the Deliverable Results; A role defined to represent management outside the Team. In practice the Business Owner is either the 'lead' Stakeholder, the Team's Sponsor, or the Product Owner's Product Owner.[25]

[22] ISO/IEC/IEEE 42010. (2011). *Systems and Software Engineering — Architecture: A Conceptual Model of Architecture Description*. Retrieved August 11, 2021, from http://www.iso-architecture.org/ieee-1471/cm/.
[23] Wikipedia contributors. (2020, March 20). *Architectural pattern*. Wikipedia. https://en.wikipedia.org/wiki/Architectural_pattern.
[24] Information Technology Infrastructure Library (ITIL). IT Service Management and the IT Infrastructure Library (ITIL). IT Infrastructure Library (ITIL) at the University of Utah. Retrieved June 15, 2021, from https://itil.it.utah.edu/index.html.
[25] Scrum Dictionary. Business Owner. ScrumDictionary.Com. Retrieved April 17, 2021, from https://scrumdictionary.com/term/business-owner/.

**Control (*v*)**

(*regulate*) to exercise restraining or directing influence over; (*reduction*) to reduce the incidence or severity of especially to innocuous levels; (*govern*) to check, test, or verify by evidence or experiments

Merriam-Webster (URL)

The means of managing risk, including policies, procedures, guidelines, practices or organizational structures, which can be of an administrative, technical, management, or legal nature. Scope Notes: Also used as a synonym for safeguard or countermeasure. See also Internal control.[26]

**Cache**

Caching is a requirement for building scalable microservice applications. Data can be cached in memory or on fast local disks.

**Control Objective**

A statement of the desired result or purpose to be achieved by implementing control procedures in a particular process.[27]

**Control Framework**

A set of fundamental controls that facilitates the discharge of business process owner responsibilities to prevent financial or information loss in an enterprise.[28]

**Developer**

A business or technology professional that builds software programs; a computer programmer (syn.) can refer to a specialist in one area of computers, or to a generalist who writes code for many kinds of software in one or more computer programming languages.[29]

**Enterprise Operator**

The individual or organization responsible for strategic design recommendations. They determine, by applying their knowledge of cloud, container and microservices components to the problems of the business; the best architecture to meet the strategic needs of the business. Additionally, they develop and maintain solution roadmaps and oversee their adoption working with developers and operators to ensure an efficient and effective solution implementation.

[26] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.
[27] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.
[28] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.
[29] Wikipedia contributors. (2021b, August 7). *Programmer*. Wikipedia. https://en.wikipedia.org/wiki/Programmer.

**Inter-Mediation**

An API Facade is a layer or gateway that sits between the microservices and the API exposed to external services. The facade creates a buffer or layer between the interface exposed to apps and app developers and the complex services. You may have several API's into different microservices, the facade abstracts the complexity with a simple singular interface.

**Microservice Architectural Style**

A microservices architecture usually refers to an application that has been structured to use basic elements called microservices, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.[30]

**Operator**

The individual or organization responsible for the set of processes to deploy and manage IT services. They ensure the smooth functioning of the infrastructure and operational environments that support application deployment to internal and external customers, including the network infrastructure, server and device management, computer operations, IT infrastructure library (ITIL) management, and help desk services.[31]

**Product Owner**

The person who identifies the customer need and the larger business objectives that a product or feature will fulfill, articulates what success looks like for a product, and drives a team to turn product vision into a reality. [32, 33]

**Propagation**

Propagation refers to the propagation of a security context through different services.

[30] Cloud Security Alliance. *Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems* (Cloud Security Alliance: 2019) 42
[31] Cloud Security Alliance. *Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems* (Cloud Security Alliance: 2019) 42
[32] Mansour, S. (2020). *Product Manager.* Atlassian Software. https://www.atlassian.com/agile/product-management/product-manager.
[33] Agile Alliance. *Product Owner.* Accessed August 10, 2021 at https://www.agilealliance.org/glossary/product-owner/.

**Software**

A collection of data or computer instructions that tell the computer how to work. Physical hardware, from which the system is built, performs the work.[34]

**Software Architecture**

The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.[35]

**Software Design Pattern**

A general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations.[36]

**Solution**

A solution is the application of architecture, patterns, and design effort  to solve a specific industry need or business problem. A solution intends to provide ongoing customer and business owner value.

**Security Policy**

A high-level document representing an enterprise's information security philosophy and commitment.[37]

**Security Procedure**
The formal documentation of operational steps and processes that specify how security goals and objectives set forward in the security policy and standards are to be achieved.[38]

[34] Cambridge Dictionary. (2021, August 11). *Software*. https://dictionary.cambridge.org/dictionary/english/software.
[35] Bass, L., Clements, P. C., & Kazman, R. (2012, September). *Software Architecture in Practice, Third Edition*. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264.
[36] Wikipedia contributors. (2021a, June 14). *Software design pattern*. Wikipedia. https://en.wikipedia.org/wiki/Software_design_pattern
[37] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.
[38] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.

**Security Standard**

Practices, directives, guidelines, principles or baselines that state what needs to be done and focus areas of current relevance and concern; they are a translation of issues already mentioned in the security policy.[39]

**Security Testing**

Ensuring that the modified or new system includes appropriate controls and does not introduce any security holes that might compromise other systems or misuses of the system or its' information.[40]

**Security Architecture**

represents the portion of the enterprise architecture that specifically addresses information system resilience and provides architectural information for the implementation of capabilities to meet security requirements.[41]

**Security Controls Overlay**

An overlay is a fully-specified set of controls, control enhancements, and supplemental guidance derived from the application of tailoring guidance to control baselines.[42]
For more information about Control Overlays, NIST Special Publication NIST SP 800-53 Rev 4., Section 3.3 Creating Overlays, and Appendix I, Overlay Template.

**Strangle**

A "Strangler" is a reference model that is used to describe the process of modernizing a monolithic application into a microservices architecture, by adding new microservices to the application over time, while decommissioning certain features of the monolith over time. It is a dissect and transition as you develop on the go model.

**Technical Debt**

A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).[43]

[39] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.
[40] ISACA. *Interactive Glossary & Term Translations*. Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary.
[41] Gantz, S. D., & Philpott, D. R. (2013). *FISMA and the Risk Management Framework*. ScienceDirect.
[42] NIST Information Technology Laboratory: Computer Security Resource Center (CRSC). (2009, June 12). *FISMA Implementation Project*. https://www.nist.gov/programs-projects/federal-information-security-management-act-fisma-implementation-project.
[43] McConnell, S. (2013). "Managing Technical Debt (slides)," in Workshop on Managing Technical Debt (part of ICSE 2013): IEEE, 2013.

**Transform**

Transformation of data implies extracting data from a source, transforming it or converting it to one format or another, and loading it into a target system.

**Translate**

An adapter microservice wraps and translates (usually function based) services into an entity-based REST interface. This allows an interface of an existing class to be used as another interface.

**Utility**

Sidecar mesh abstracts the underlying infrastructure through a proxy of services below the application. The proxy handles traffic flow, inter-microservice communication, connection, management, load balancing, availability and telemetry data. The sidecar mesh paradigm provides orchestration and architectural independence from underlying cloud architectures, across multiple clouds.

# Appendix C: References

Following is a table of references used throughout this paper.

| | |
|---|---|
| Bass, L., Clements, P.C., & Kazman, R. (2012, September). | Bass, L., Clements, P. C., & Kazman, R. (2012, September). Software Architecture in Practice, Third Edition. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264. |
| Boehm, B. & Basili, V. R. (2001). | Boehm, B., & Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Computer*, 34(1), 135–137. |
| Cloud Security Alliance. (2019, July 8). | Cloud Security Alliance. (2019, July 8). *Six Pillars of DevSecOps*. https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/. 5-6 |
| Cloud Security Alliance. (2019, August 1). | Cloud Security Alliance. (2019, August 1). *Information Security Management through Reflexive Security.* https://cloudsecurityalliance.org/artifacts/information-security-management-through-reflexive-security/. (13, 14, 16) |
| Cloud Security Alliance. (2020, February 24). | Cloud Security Alliance. (2020, February 24). *Best Practices in Implementing a Secure Microservices Architecture.* https://cloudsecurityalliance.org/artifacts/best-practices-in-implementing-a-secure-microservices-architecture/. |
| Docker. (2021). | Docker (2021). *Docker Hub: Set Up Automated Builds*. Docker. https://docs.docker.com/docker-hub/builds/ |
| Enterprise Integration Patterns. (2021). | Enterprise Integration Patterns. *Enterprise Integration Patterns - Messaging Patterns Overview.* Retrieved August 11, 2021, from https://www.enterpriseintegrationpatterns.com/patterns/messaging/ |
| Github. (2019). | Github. (2019.) *Guide on Microservices: Backups and Consistency \| Consistent Disaster Recovery for Microservices - the BAC Theorem.* dhana-git/Guide on Microservices: Backups and Consistency.md. Retrieved August 11, 2021, from https://gist.github.com/dhana-git/3dda5326b3bd15a93d3389a6c30d3000. |
| Hinkley, C. (2019). | Hinkley, C. (2019, November 6). *Dissecting the Risks and Benefits of Microservice Architecture*. TechZone360. https://www.techzone360.com/topics/techzone/articles/2019/11/06/443660-dissecting-risks-benefits-microservice-architecture.htm. |
| IBM. (2021). | IBM. *Patterns in Event-Driven Architectures – Introduction.* IBM Garage Event-Driven Reference Architecture. Retrieved August 11, 2021, from https://ibm-cloud-architecture.github.io/refarch-eda/patterns/intro |

| | |
|---|---|
| Microsoft. (2021). | Microsoft. (2021). Architecture Best Practices: Caching. Microsoft https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching |
| NIST. (2015, January 22). | NIST. (2015, January 22). *NIST Special Publication 800-53, Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations.* https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final |
| NIST. (2020, September). | NIST. (2020, September). *NIST Special Publication 800-53, Revision 5: Security and Privacy Controls for Information Systems and Organizations.* https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf |
| NIST. (2021, August 4). | NIST. (2021, August 4). *Security and Private Control Overlay Overview.* Retrieved August 11, 2021, from https://csrc.nist.gov/projects/risk-management/sp800-53-controls/overlay-repository/overlay-overview. |
| Pardon, G., Pautasso, C., & Zimmerman, O. (2019). | Pardon, G., Pautasso, C., & Zimmerman, O. (2019). *Consistent Disaster Recovery for Microservices: The BAC Theorem*. IEEE Cloud Computing. https://design.inf.usi.ch/sites/default/files/biblio/bac-theorem.pdf |
| Raible, M. (2020). | Raible, M (2020). *Security Patterns for Microservice Architectures, Encryption and Protection Secrets*. Okta. https://developer.okta.com/blog/2020/03/23/microservice-security-patterns#5-encrypt-and-protect-secrets |
| Rance, S. (2017, June 22). | Rance, S. (2017, June 22). *How to Define, Measure, and Report IT Service Availability*. ITSM Tools. https://itsm.tools/how-to-define-measure-and-report-service-availability/. |
| The Open Group. (2021). | The Open Group. *The TOGAF Standard, Version 9.2 Overview, Phase A, B, C, D, E, F, and G*. Retrieved August 11, 2021, from https://www.opengroup.org/togaf. |
| Wiggins, A. (2017). | Wiggins, A. (2017). The Twelve-Factor App. https://12factor.net/ |
| Agile Alliance. (2021). | Agile Alliance. *Product Owner.* Accessed August 10, 2021 at https://www.agilealliance.org/glossary/product-owner/ |
| Bass, L., Clements, P.C, & Kazman, R. (2012, September). | Bass, L., Clements, P. C., & Kazman, R. (2012, September). *Software Architecture in Practice, Third Edition*. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30264. |
| Cambridge Dictionary. (2012, August 11). | Cambridge Dictionary. (2021, August 11). *Software*. https://dictionary.cambridge.org/dictionary/english/software |

| | |
|---|---|
| Cloud Security Alliance. (2019). | Cloud Security Alliance. *Challenges in Securing Application Containers and Microservices Integrating Application Container Security Considerations into the Engineering of Trustworthy Secure Systems* (Cloud Security Alliance: 2019) 42. |
| Gantz, S. D., & Philpott, D. R. (2013). | Gantz, S. D., & Philpott, D. R. (2013). *FISMA and the Risk Management Framework.* ScienceDirect. |
| Information Technology Infrastructure Library. (2021). | Information Technology Infrastructure Library (ITIL). IT Service Management and the IT Infrastructure Library (ITIL). IT Infrastructure Library (ITIL) at the University of Utah. Retrieved June 15, 2021, from https://itil.it.utah.edu/index.html |
| ISO/IEC/IEEE 42010. (2011). | ISO/IEC/IEEE 42010. (2011). *Systems and Software Engineering — Architecture: A Concep ISACA. Interactive Glossary & Term Translations.* Retrieved August 11, 2021, from https://www.isaca.org/resources/glossary *Model of Architecture Description*. Retrieved August 11, 2021, from http://www.iso-architecture.org/ieee-1471/cm/ |
| Mansour, S. (2020). | Mansour, S. (2020). Product Manager. Atlassian Software. https://www.atlassian.com/agile/product-management/product-manager |
| McConnell, S. (2013). | McConnell, S. "Managing Technical Debt (slides)," in Workshop on Managing Technical Debt (part of ICSE 2013): IEEE, 2013. |
| NIST Information Technology Laboratory. (2009, June 12). | NIST Information Technology Laboratory: Computer Security Resource Center (CRSC). (2009, June 12). *FISMA Implementation Project.* https://www.nist.gov/programs-projects/federal-information-security-management-act-fisma-implementation-project |
| Scrum Dictionary. (2021). | Scrum Dictionary. Business Owner. ScrumDictionary.Com. Retrieved April 17, 2021, from https://scrumdictionary.com/term/business-owner/ |
| Wikipedia contributors. (2020). | Wikipedia contributors. (2020, March 20). *Architectural Pattern.* Wikipedia. https://en.wikipedia.org/wiki/Architectural_pattern |
| Wikipedia contributors. (2021, August 7). | Wikipedia contributors. (2021b, August 7). *Programmer*. Wikipedia. https://en.wikipedia.org/wiki/Programmer |
| Wikipedia contributors. (2021, June 14). | Wikipedia contributors. (2021a, June 14). *Software design pattern.* Wikipedia. https://en.wikipedia.org/wiki/Software_design_pattern |

# Appendix D: Work Instruction

## 1.0 Microservice Architecture Pattern Template

### 1.1 Pattern Work Instruction

Please align your patterns to the MAP reference architecture.  Create a pattern using the format and instructions below.

### 1.2 Pattern Template

| **Software Pattern Name <name> [Template]** | |
|---|---|
| **Version** | <number 0.0><br>The latest iteration of the pattern + overlay |
| **Pattern Purpose** | <purpose><br>What is the purpose of the pattern? What is it supposed to do? |
| **Plane Location** | <platform plane/ software plane/ hybrid-distributed><br>Which Plane or Planes? Does the pattern exist in the platform plane, the software plane, or span both? |
| **Structural Description** | <It does what?><br>Describe what it does - SIPOC - Source, Input, Process, Output, Consumer |
| **Behavior Description** | <It behaves how?><br>Describe the behaviour under normal circumstances |
| **Data Disposition** | <Uses data / transits data / data at rest><br>Does it make new data, transit existing data, or store data at rest? |
| **Major Dependencies** | <without it, the pattern breaks><br>Is it tightly bound to the nominal function or something else? |
| **Minor Dependencies** | <without it, the pattern performs sub nominally><br>What platform or software plane element has to perform poorly for the pattern to perform sub-optimally |
| **Internal Event/ Messaging Needs** | <any needed inter-process or inter-pattern communication?><br>Is nominal function dependent on internal messaging from another pattern? |

| | |
|---|---|
| **External Event/ Messaging Linkage** | <what for or type of logging - AppDev Debug, SecOps, DBtransact> Does activity need to be logged somewhere and for what purpose? |
| **Event Response Behavior** | <when an event condition occurs, what happens?> What is the response behavior to DOS or degraded performance? |
| **Common Upstream Linkage** | <any linkages to other patterns?> Are there any commonly linked upstream patterns? |
| **Common Downstream Linkage** | <any linkages to other patterns?> Are there any commonly linked downstream patterns? |
| **Ops Security Tie-Back** | <IT Security control(s)?> (infrastructure As code and configuration> Overlay Step 1: What IT Security Controls apply? |
| **DevSecOps Tie-back** | <Application security control(s)?> (DEV Pipeline SAST and DAST?> Overlay Step 2: What Application Security Controls apply? |
| **Evaluation Method** | <verification of controls method> Overlay Step 3: What are acceptable control verification methods? |
| **Control Overlay Disposition** | <preventative/ corrective/ detective> Overlay Step 4: Is the control(s) preventative, corrective, or detective? |
| **Composite Status (unique/ generalized)** | <generalized, same control overlay as [pattern]; Has unique control overlay> Can the overlay be generalized to other patterns, or is it unique to this pattern? |

# 2.0 Security Overlay Template

## 2.1 Security Overlay Work Instruction

After you create a pattern in accordance with section 5 above the next step is to create a Security Overlay in accordance with the instructions below.

### 2.1.1 Introduction

The primary goal of the work instruction is to provide guidance and standardize the approach to developing the security control overlay for the architectural patterns found in the CSA MAP (Microservices Application Pattern) paper. Although the content can aid any number of roles, it is tailored to security architecture, the MAP paper specifically, and the architect's point of view. The secondary work instruction goal is to create a non-statistical decomposition/re-composition approach to security architecture that remains economical and lightweight without sacrificing the basics of control selection in the design process. The method seeks to avoid the abstract and sometimes esoteric risk conversation that occurs in control conversations. The method contains 8 steps that render a software architecture patterns' security controls overlay.

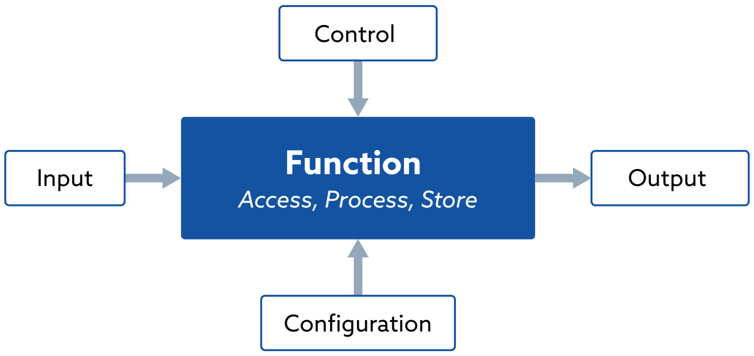
Expect control overlap with other patterns. Reduction of controls occurs by comparing all other pattern overlays generated through this method. Although the method is prone to subjective professional judgment, its limitations outweigh its benefits. The method standardizes development of security control overlays for software architectural patterns found in microservice applications; it is tailored specifically to serve as a work instruction for the CSA Map paper.

The method is transferable to other situations and circumstances where the risk appetite, control work set, and industry are different. The method can be prefixed as a preliminary to a formal risk assessment and control baseline determination. The MAP pattern overlay method is not a substitute for *NIST Special Publication SP800-53 version 4, "Security and Privacy Controls for Federal Information Systems and Organizations", Chapter 3, "The Process" (of selecting security control baselines) pages 28 through 44, or NIST SP800-53 Appendix D (beginning on page D-1).*

### 2.1.2 Preliminaries

The work instruction uses specific terms and phrases as follows:

1. CONTROL (disposition): Controls classify as behavioral/functional (e.g. this control functions to prevent denial of service), or classify by nature/type (e.g. this control requires two physical signatures to prevent falsification of paper documents)
    a. Preventive, detective, and corrective controls are behavior/functional controls. Descriptions align with the timing of the actions performed.
    b. Physical, administrative, technical, and regulatory controls are nature/type controls. Descriptions align with the type of action performed.
    c. Control disposition can be further sub-classed as "feed-forward" (anticipate problems), "Concurrent" (correct as problems occur), and "feedback" (Correct problems after occurence).

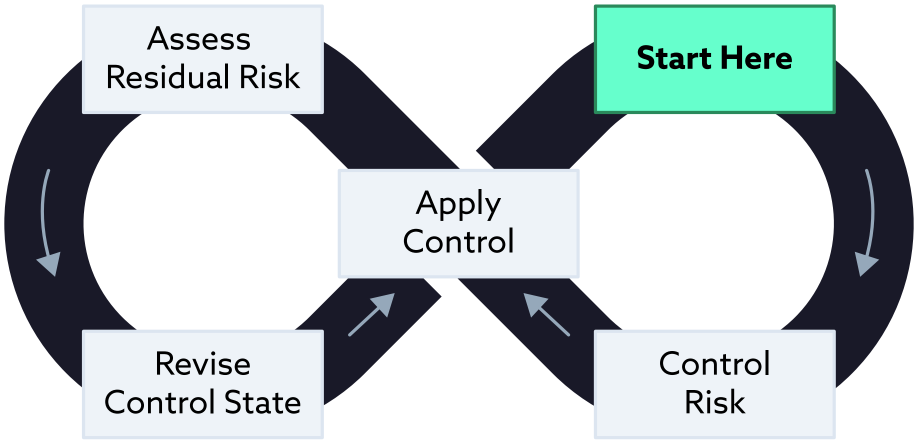


2. DISPOSITION: A quality, character, tendency, of something to act in a certain manner under given circumstances.
3. EFFECTIVENESS (as an architectural quality meta-attribute): Control effectiveness can be appraised through the selection of quality attributes. Applying controls is a means to manage according to a defined organizational risk appetite. As a security architect, the top-line goals in design are,
    - Monitor performance and take action to ensure a desired result,
    - Increase the probability that actions occur as intended, and with the correct timing

Security control architecture attributes are,

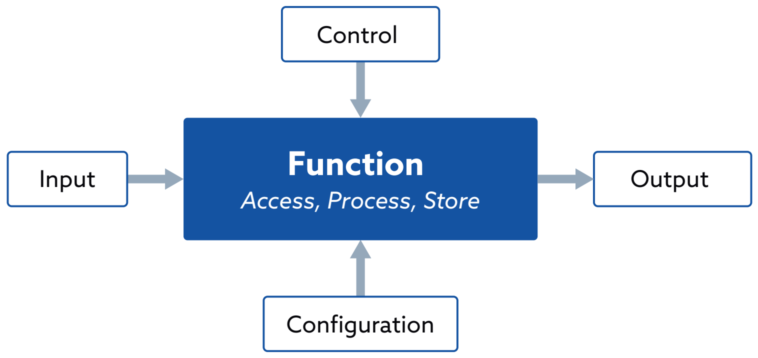
a. Accuracy (coarse-grained or fine-grained action)
b. Chronology (Timing of control action)
c. Economy (total cost of control)
d. Flexibility (threat and/or use case coverage)
e. Understandability (comprehend-able actions and outcomes)
f. Reasonability (balance between performance and security)
g. Criteria applied (Parameterized tuning of control action)
h. Impact (Risk-reward trade-off or control application)

4. BUILDING CONTROL OVERLAYS When assessing a MAP reference architecture patterns' need for a control overlay, the opening conversation must be about data risk. Discuss threats to confidentiality, integrity, and availability. Begin with a risk assessment conversation assessing the risks (yellow below). Representation of an end-to-end risk conversation begins with assessing risk and proceeds as follows,

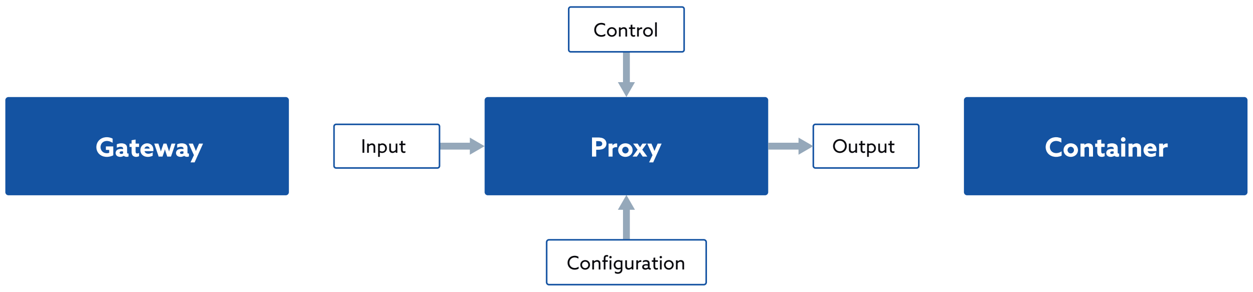


### 2.1.3 Steps

1. Refer back to the generic control state diagram. Consider all data in use assigned a data classification of "private-internal" and existing in a LOW risk baseline assignment as per NIST Special Publication SP800-53 version 4, "Security and Privacy Controls for Federal Information Systems and Organizations", Appendix D (beginning on page D-1). Example -



2. Using the MAP Reference architecture, step back and look at the inputs to the pattern, and the outputs of the pattern , assess the needed configuration. Return to the generic control state diagram and re-label the inputs, function (pattern name), and outputs taken from the MAP reference architecture. Example -



3. Prepare an operation-centric point of view using a matrix.  Interrogate the updated control state diagram and ask "Who (can access)," "What (can access)," "When (can it be accessed – under what conditions?), "How (can access occur)," and "Where (can access occur)." Example - Who (can access the function)? Upstream interface restricted by ACL, or mTLS certificate What (can access the function)? Other compute interfaces When (can the function be accessed / conditions)? At runtime, through configuration, and in QA and TEST ENV. How (can function access occur)? Through authentication (authN)and authorization (authZ) Where (can function access occur)? In the production, QA, and Test environments; via port and protocol, via intermediary capability. For the purposes of this exercise, consider configuration as part of the operations point of view.  Select controls that target a secure configuration.

| | |
|---|---|
| *Who (can access the function)?* | Upstream interface restricted by ACL, or mTLS certificate |
| *What (can access the function)?* | Other compute interfaces |
| *When (can the function be accessed/ conditions)?* | At runtime, through configuration, and in QA and TEST ENV. |
| *How (can function access occur)?* | Through authentication (authN)and authorization (authZ) |
| *Where (can function access occur)?* | In the production, QA, and Test environments; via port and protocol, via intermediary capability. |

4. Prepare a data-centric point of view using a matrix. Ask the same questions as outlined in step 3, but change "can" to "can't." Defaulting to deny in the viewpoint interrogation is the basis for "need to know" and the first building block of "defense in depth." The sensitivity of the data being accessed, processed, or stored is directly proportional to control state depth. Example -

| | |
|---|---|
| *Who (cannot access the function)?* | Developers, Operators, Architects, Tech Support, Manager, non-employees |
| *What (cannot access the function)?* | Interactive command shell; ad-hoc scripted commands |
| *When (is the function not accessible/ conditions)?* | At runtime, in production when container hosted, via ad-hoc scripting |
| *How (can function access be prevented)?* | Input filters, file system level controls, port and protocol restrictions |
| *Where (can function access be denied)?* | Upstream, downstream, configuration, port and protocol restrictions |

5. Prepare a people-centric point of view using a matrix. Ask the same question in step 4, but apply human access as the lens through which the control state interrogation occurs. Example -

| | |
|---|---|
| *Who (cannot access the function)?* | DEV and OPS denied production and runtime access; DEV and OPS restricted access to QA. |
| *What (cannot access the function)?* | Developer command shell; ad-hoc scripted commands |
| *When (is the function not accessible/ conditions)?* | Outside business hours, excluding for administrative duties |
| *How (can function access be prevented)?* | AuthN, AuthZ, File system level controls, port and protocol restrictions, ACL, xBAC (Role, Context, Attribute Policy) |
| *Where (can function access be denied)?* | Upstream, downstream, via configuration restrictions, access inter-mediation, port and protocol usage. |

6. Return to the risk assessment commentary. Now that four different views of control state diagram exist, apply all to form a risk assessment. Do the risk assessment activity iteratively until satisfied that the pattern control state meets the need. Example -

| | |
|---|---|
| *Assess the risk (what are the risk types?)* | Access risk, Transmission disclosure risk, Service denial risk |
| *Control the risk (what are the main control categories)* | Access controls, transmission controls, resiliency controls |
| *Apply the controls (Where?)* | Upstream platform, runtime configuration, to the function itself. |
| *Assess residual risk (Is it unacceptable data risk)* | Insider threats, novel software vulnerabilities, malfunction |
| *Revision of control state (improvements?)* | Reapply additional controls to mitigate residual risk |

7. Use the three matrices and the risk assessment, select control families from the *NIST Special Publication SP800-53 version 4 Security and Privacy Controls for Federal Information Systems and Organizations Section 2.2 Security Control Structure Page 9* (Do not select from the individual control). In order to perform step 7, familiarity with the publication contents is a prerequisite. Security architecture requires the practitioner to be very familiar with many industry standard control work sets. Example -

| **Control Category** | **NIST SP 800-53 Family** | **Control Disposition** | **Plane Affinity Inheritability** |
|---|---|---|---|
| *Access controls* | *AC- Access Control* | *Preventative* | *Enterprise plane-inheritable* |
| *Transmission controls* | *SC- System Communications and Protection* | *Primarily detective, secondarily preventive* | *Enterprise plane-inheritable* |
| *Configuration Controls* | *CM- Configuration Management* | *Primarily detective, secondarily preventive* | *Enterprise plane-inheritable* |
| *Software assurance controls* | *SA- System and Service Acquisition* | *Preventive* | *Software plane-unique and inheritable* |

8. In step 8, decompose control family selection in step 7 by selecting the individual controls from within the *NIST Special Publication SP800-53 version 4 Security and Privacy Controls for Federal Information Systems and Organizations*[44] according to the control families listed. Re-compose step 7 with individual controls assigned by family, related by control type, and plane affinity. After completion, the control overlay for the software architecture pattern concludes. *Use NIST SP800-53 Appendix D (beginning on page D-1) to aid this step. Remember the information classification is LOW.* Note, no formula for exact selection exists, control selection varies by risk appetite.

[44] https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final