*Part* 1 2 3 4 **5**

Introduction to Software Architecture with Actors:  
Part 5 – On Fragmented Systems

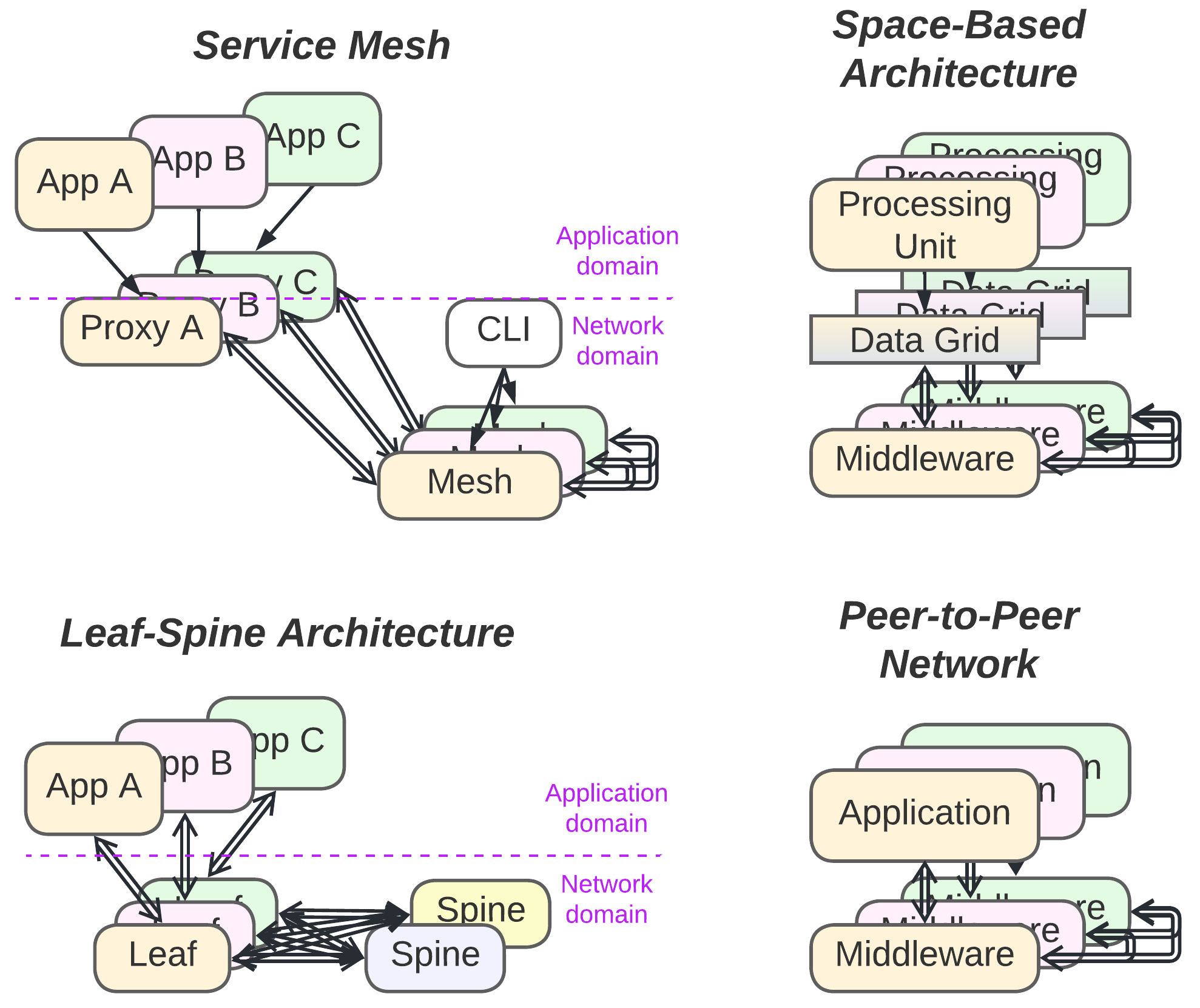
The previous part of this cycle was dedicated to systems with *models*, i.e. horizontal layers that spread through an entire domain. If a structural diagram for a system is drawn with abstraction increasing upwards, the model may be found above services (resulting in a Π-shaped system), below them (U-shaped) or in the middle (H-shaped). For each type of system structure, a set of three distinct patterns emerges through variations in the distribution of the business logic over the components.

The preceding article described the three basic ways to divide a monolith, namely: sharding (spawning identical instances), layers (splitting by abstraction) and services (splitting by subdomain).

Now, according to the [*Rule-of-Three*](https://en.wikipedia.org/wiki/Planescape#Rule-of-Three), there should be three kinds of fragmented architectures with neither monolithic layers (models) nor non-layered subdomains.

## **Mesh**

* + 1. Cutting a monolith along all the axes under discussion (*abstraction*, *subdomain*, *sharding* – see Part 3) simultaneously does not make much sense, as the resulting structure is too hard to manage, that is, unless it relies on communication between shards to form a *mesh*.



***A customizable distributed middleware for unreliable networks***. The lower layer of services implements network connectivity, while the upper services contain the business logic. Two domains, namely, *application* and *network*, are present.

The *network* domain may supply *proxies* that virtualize network resources for the *applications*, but is generally built around *mesh* shards that sustain network topology and are usually managed via a global *configurator*. *Proxies* and *mesh* *actors* have different properties; the *proxies* provide a convenient connectivity and cached resource abstraction to the *applications*, which usually includes (some of) the: blocking calls, retries, virtual connection objects and *data grid*, while *mesh* is occupied with a great deal of the low-level, and often hardware-dependent, real-time communication.

The other domain is the *application* level, where different kinds of business *applications* run on top of and communicate through identical or similar (as application protocols may vary in *Service Mesh*) *proxy* interfaces. The entire *network* layer serves as a distributed *middleware* (described in Part 4), with some implementations also providing a *shared repository* (also from Part 4).

In this type of system, all the components have specific roles and are usually developed and deployed independently. This explains the granularity observed in the structure. However, from the viewpoint of business logic, it is mostly similar to a system of *services* or *shards* (both described in Part 3) that make up the application level; thus, the entire network layer tends to be excluded from architectural discussions once implemented or purchased.

*Benefits* (in addition to those of *Services*, Part 3):

* The system tolerates the loss of any component thanks to its decentralization and redundancy; this way, it can run on commodity hardware or even use the leftovers of CPU time at volunteers’ desktops worldwide.
* The extra layer of *proxies* allows for abstracting distributed resources, e.g. providing a virtual view of physically distributed data in *Space-Based Architecture* and torrents.
* Perfect scalability is achieved.
* The unified transport interface simplifies development.

*Drawbacks* (in addition to those of *Services*, Part 3):

* Scenarios that involve multiple *applications* may be tremendously hard to debug due to unstable failures at the network level.
* Scenarios that involve multiple *applications* may be unstable if *Mesh* is run over an unreliable network or low-end hardware; therefore, shards of a single *application* type must be interchangeable (either stateless or with replicated data) to provide redundant service capacity.
* There is a high communication overhead that causes delays in messaging and slows down access to distributed resources for systems not limited to a single data center.
* Use cases may need to retry on failures at any step and tolerate lost or duplicated messages, cases of multiple diverging responses to a single request included.
* The transport layer is complex; thus, an out-of-the-box product should be used wherever possible.
* The unified transport interface may be non-optimal for some use cases.

*Summary*: *Mesh* provides a shared virtual space over unreliable networks, often at the expense of communication speed and stable participants’ identities.

*Common names*: Peer-to-peer / Ad-hoc Networks*.*

*System architecture*: [**Service Mesh**](https://buoyant.io/service-mesh-manifesto), Space-Based Architecture [[SAP](#kix.5yhcd6l42sgh)].

Real-world applications of *Mesh* architectures include data center infrastructure, torrents and even [finances](https://jimmysongio.medium.com/introducing-sofamesh-a-solution-for-large-scale-service-mesh-based-on-istio-by-ant-financial-84ff51fd66f6). Like *Middleware* and *Microkernel* (both described in Part 4), *Mesh* is also usually dropped from architectural diagrams, as it is transparent for business logic.

There are many variants of meshes designed for different uses. I will provide short descriptions for some of the ones that I find interesting:

### [**Service Mesh**](https://buoyant.io/service-mesh-manifesto)

Proxies may contain application-specific code that handles protocol format conversions (e.g. JSON <-> XML <-> Protobuf) and implements aspects (security, observability). This provides an extra layer of indirection, with independent development and deployment, between the business logic and the network implementation.

*Service Mesh* frameworks are usually purchased to kickstart projects that implement *Microservices* (Part 3) or to merge systems that have been developed independently.

### [**Leaf-Spine Architecture**](https://www.geeksforgeeks.org/spine-leaf-architecture/)

This is a two-layer *full mesh* (where every device in one layer is directly connected to *all* of the devices in the other layer) for use inside data centers. The full connectivity approach both guarantees that the failure of any one device will not impair the network speed for other components and provides a high data transfer rate with commodity hardware. However, the size of the mesh is limited by the number of physical ports in the network switches used, and wiring the thing involves hundreds or thousands of network cables.

*Leaf-Spine Architecture* provides fast, cheap and stable connectivity between servers within data centers of moderate size.

### **Peer-to-Peer Network**

Here, all the *nodes* are equal and able to find each other. Moreover, *proxy* and *mesh* components are merged together to optimize the single task the *P2P Network* is specialized in.

*P2P Networks* are used for fault tolerant communication and data exchange (torrent, onion, bitcoin).

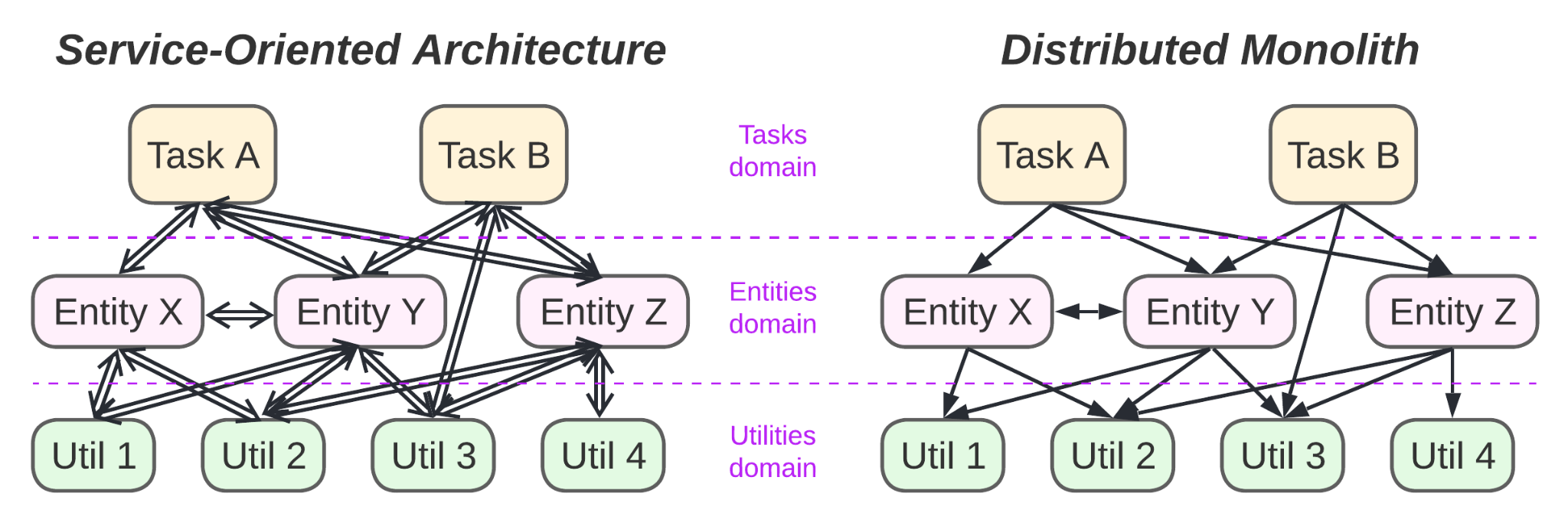
### **Space-Based Architecture** [[SAP](#kix.5yhcd6l42sgh)]

This pattern seems to be a *P2P Network* featuring a *data grid* – a virtualized *shared repository* (from Part 4) that provides every node’s *application* with access to the entire project’s data, which is distributed and replicated over multiple shards. It works like the OS’s swap files; if the *application* (called the *processing unit*) needs to access a piece of data that is not available in the node’s RAM, the proxy (*data grid*) blocks the *application* and sends a request to its peers to get a copy (and possibly ownership) of the data. As soon as the data has been delivered and unpacked to the node’s RAM, the waiting *processing unit* thread is notified and proceeds as if the data originally belonged to the node. The *data grid* tracks ownership for data segments (to avoid write conflicts) and takes care of replication and persistence (to provide parallel read access and avoid data loss). It is also likely to contain some logic that assigns user requests to nodes which already have the required data in RAM (improving temporary data access locality).

*Space-Based Architecture* can be used for projects that work with huge datasets and are deployed over multiple data centers if the actions that run on the dataset are local (i.e. do not require reading through or transforming much of the dataset).

Here, once again, a family of related distributed implementations (*Meshes*) targets two architectural patterns (namely, *Middleware* and *Shared Repository*, both described in Part 4) that have similar structural diagrams, proving that a pattern’s structure defines its properties and thus its relations to other architectures.

## **Distributed Modules (SOA)**

* + 1. The preceding structure (*Mesh*) had deep sharding, explicit layers (application, proxy, connectivity) and subdomains (application support and network topology), and few component types. If those restrictions are lifted, the [chaos](http://www.soa-manifesto.org/default.html) emerges:
    2. 

***Sacrificing everything for modularity***. If the modules of a *monolith* (Part 2) are distributed over a network, the result is, surprisingly, a *distributed monolith* [[MP](#kix.wzzg9asvgv6o)] for synchronous RPC calls or a *service-oriented architecture* (*SOA*) for asynchronous communication.

The diagrams omit a unified *middleware* (*Enterprise Service Bus*) that connects and manages the *services*.

*Benefits*:

* The strongest possible reduction in *dev* (module code) complexity (see Part 1) is obtained through the division of the business logic along both the *abstraction* and *subdomain* axes (refer to Part 3 for the system of coordinates analyzed).
* Each module may run on a dedicated hardware that is suitable to its needs.
* *Asynchronous* systems are able to survive the failures of individual modules.
* The modules are sharded and deployed independently.

*Drawbacks*:

* *Ops* (module integration) complexity is high, as the modules depend on each other’s interfaces and contracts.
* The interdependencies of the modules cause interdependencies among the teams that develop those modules.
* With *synchronous* communication, a single failed low-level module may halt the entire system.
* All the use cases are slow because too much messaging is involved.
* All the use cases are nearly impossible to debug because of the number of distributed components involved.

*Evolution*:

* If the integration complexity hits hard or the performance suffers from having too much distributed communication, merging the services into coarse-grained components should be considered. The options include *Monolith* (Part 2) or *Layers* (Part 3) for smaller projects (a tremendously rare case with *SOA*) and *(Micro-)Services* (Part 3), probably involving *Orchestrators* (Part 4), for large systems.
* If there is a need to integrate modules that use non-standard transports, a *service mesh* (described above) may be considered for use as a *middleware* (traditionally called an *Enterprise Service Bus* in SOA).

*Summary*: *SOA* decreases the individual modules’ code complexity without introducing any code or data duplication among the modules at the cost of incurring overwhelming integration complexity and interdependency.

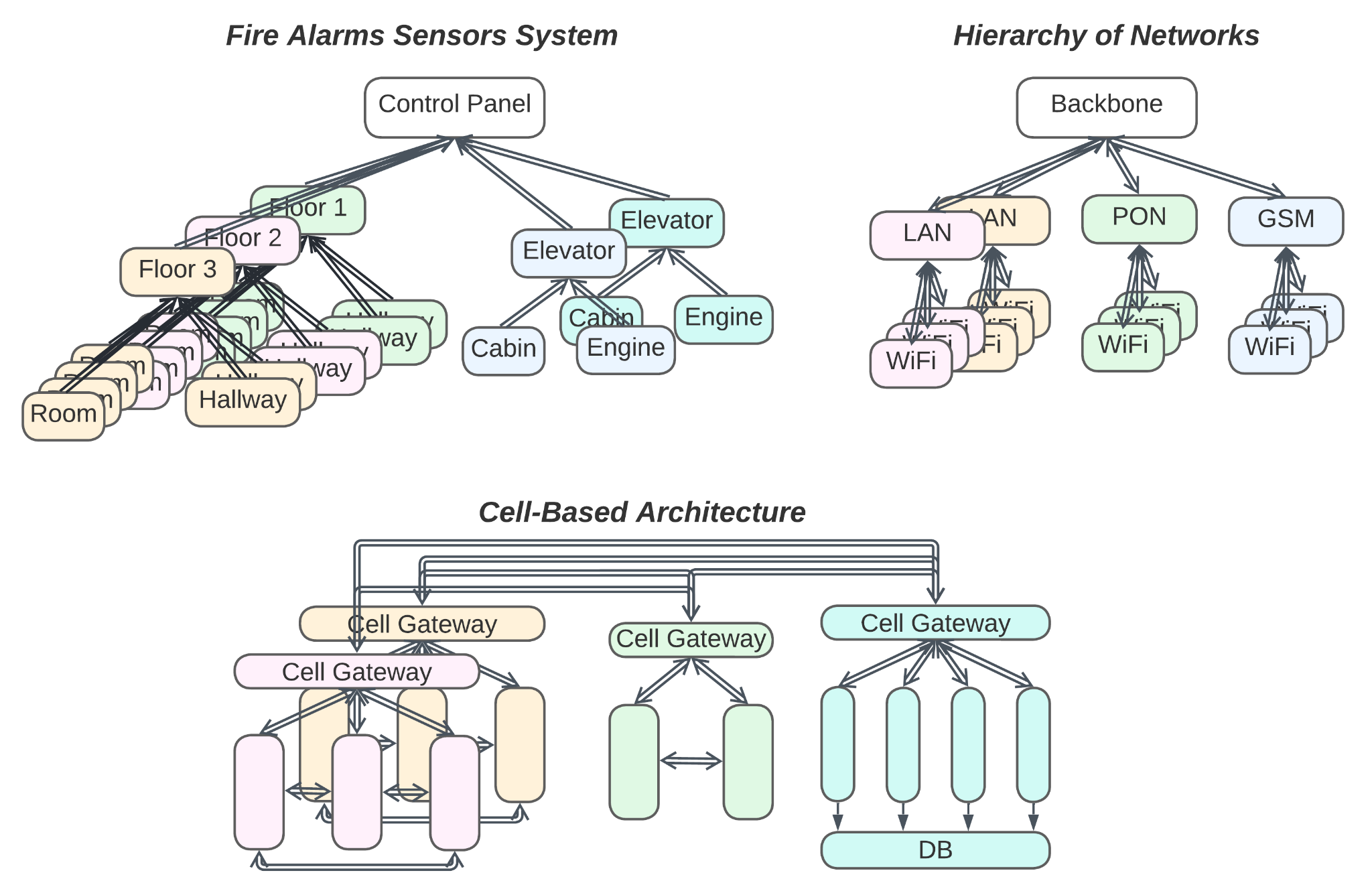
*System architecture*: [**Service Oriented Architecture**](https://patterns.arcitura.com/soa-patterns/basics/soamethodology/service_layers),Distributed Monolith (synchronous RPC calls) [[MP](#kix.wzzg9asvgv6o)].

This approach should have worked in an ideal world with perfect interteam communication and zero-cost distributability. Nevertheless, the method survived, mostly in automotive, avionics and legacy enterprises (that had put lots of resources into implementing *SOA* while it was considered a brave new fashion).

Why is *SOA* still thriving in vehicle engineering? Probably because it allows the services to be dispersed over cheap, spatially distributed chips. If there are already physical black boxes being used for logging – why not use them as system-wide logger components? A brake chip placed near a wheel does not log anything to its own flash – it sends all the logs to a remote black box. And even if the black box fails, the brake will still be functional. The reliability of the CAN bus compared to that of the Internet could also have influenced the choices made in the industry, and the same could be said of the inherent modularity of car internals, which resembles the OOP principles *SOA* was based on. Moreover, design committees in automotive, avionics and enterprise industries could well be banning any innovation for ages. Huge codebases in these domains should have been yet another driving force towards favoring *SOA* architectures.

Why did *SOA* go extinct in other kinds of backends? Because *Microservices* (detailed in Part 3) are way more independent in their development, deployment and production. This means less struggling with *ops* (i.e. deployment and integration), less inter-team synchronization, minimal downtime and better performance. It is true that with *Microservices*, the code is less granular and sometimes duplicated, but the idea of reaching high granularity was the mistake discussed in Part 1; any attempts to distribute a coupled logic or cut it with asynchronous interfaces result in slow, unstable, and unsupportable systems that are hard to understand and nearly impossible to debug.

## **Hierarchy**

* 1. The last pattern to be discussed is more of an approach than a well-defined composition of modules.
  2. 

***Using recursive modularity and polymorphism to curb complexity***. A pattern is applied recursively, often resulting in a tree-like structure with a relatively small amount of very high-level logic managing multiple, in many cases polymorphic, modules of an intermediate abstraction level, each of which supervises a set of concrete services or devices.

*Benefits*:

* The domain logic is divided into multiple, relatively small parts.
* The development, deployment, scaling and properties of the involved modules are mostly independent.
* The system is fault tolerant for any but the topmost component.
* The subsystems may serve many use cases without escalating them to the highest level.
* Local use cases tend to be fast and run in parallel.
* Both bottom-level components and entire subsystems are easily replaceable or stubbable thanks to the abundance of layers and the common use of polymorphism.

*Drawbacks*:

* The pattern is not applicable to domains that are not inherently hierarchical.
* The topmost layer may become a bottleneck for performance, stability or evolvability.
* The failure of the topmost component will halt global use cases. However, it will not influence local tasks handled by the subsystems.
* Global use cases, which spread over multiple layers, are rather slow and hard to debug.
* The start of the project may be on the slow side because many interfaces need to be settled.

*Summary*: *Hierarchy* brings many significant benefits wherever it can be used.

*Common names*: Hierarchy, Tree, Recursive Structure.

*Software architecture*: Presentation-Abstraction-Control [[POSA1](#kix.y1wcg4wyxx7o)].

*System architecture*: [**Cell-Based Architecture**](https://trmidboe.medium.com/cell-based-architecture-and-federated-microservices-4fc0cf3df5a6).

Hierarchy handles domain complexity by efficiently distributing it over multiple layers of services. If the main drawback of *Hexagonal Architecture* (Part 4) lies in its monolithic business logic layer, hierarchical *Hexagon of Hexagons* divides the business logic along both the *abstraction* and *subdomain* design dimensions. If the *Message Bus* (Part 4 – *Middleware*) logic becomes too complex, *Bus of Buses* can come to the rescue. When there are too many *(Micro-)Services* (Part 3) to integrate and deploy efficiently, *Services of Services* is the answer.

If a system looks like it can be partitioned into groups or into a coordinating component and several (preferably polymorphic) coordinated components of comparable size, it calls for the application of the *hierarchy* approach. If there is no coordinator, the system resembles *Services* (from Part 3 of this cycle). And if the coordinator seems to incorporate the largest share of the business logic, the system shifts towards Π-shaped *Hexagonal Architecture* or *Application Service* structures (both from Part 4).

The variants I am aware of include:

### **Hexagon of Hexagons (Hexagonal Hierarchy)**

The business logic of each layer operates a set of lower-level entities, each containing its own business logic. This way, the *adapters* of the topmost *hexagonal architecture* (Part 4) become *models* of the lower layer. In its simplest composition, the structure is similar to *Application Service* (Part 4) with hexagonal components.

Such structures emerge in industrial IoT (e.g. fire alarm) systems; the upper layer is a control desk with a UI, while below it reside distributed floor and elevator controllers that integrate data from individual sensors in their respective zones of control. Such multi-layer, multi-component systems are flexible enough to be easily re-configured for various kinds of buildings or updated with new sensor models.

### **Bus of Buses (Hierarchy of Networks)**

When there is a need to interconnect multiple types of networks (or *Message Buses* – Part 4), it is usually done via a single master network. Each of the lower-level networks implements a gateway, adapter or router that translates between the higher-level and lower-level networks’ protocols.

The most well-known example is the Internet or telephony networks. In the desktop world, there was the *Presentation-Abstraction-Control* [[POSA1](#kix.y1wcg4wyxx7o)] pattern, which is [likely dead by now](https://herbertograca.com/2017/08/17/mvc-and-its-variants/). It used a tree of agents (i.e. actors subscribed for external events) to present hierarchical structures with component-specific UI widgets.

### **Services of Services** ([**Cell-Based Architecture**](https://trmidboe.medium.com/cell-based-architecture-and-federated-microservices-4fc0cf3df5a6))

Having too many same-level *microservices* (Part 3) complicates integration (including tracking service dependencies) and deployment. The proposed solution clusterizes the system into comparatively strongly coupled groups. Each group gets a *gateway* (Part 4), is deployed as a whole and is treated by the other groups as a single *microservice*, notwithstanding its fragmented internal structure. This results in the deployment and integration complexity being based on the number of groups, not the number of individual *microservices*.

This structure may incorporate elements of *Hexagonal Architecture* and *Orchestrators* (both from Part 4). *Application Services* (Part 4) that feature high-level business logic may be used instead of *Gateways* if the domain allows for a partially hierarchical decomposition.

## **Design Space**

* + - 1. The *design space* [[POSA1](#kix.y1wcg4wyxx7o), [POSA5](#kr6gfasg06m)] is an [imaginary](https://youtu.be/AnaQXJmpwM4?t=22) multidimensional set that contains all the possible architectures for all kinds of systems. It is multidimensional because every architecture has multiple parameters that may be changed, each parameter serving as a dimension of the solution space.
      2. However, humans have a hard time dealing with more than three dimensions. Therefore, we need a projection in order to confine our efforts to a couple of design aspects. This is exactly what has been done over the last 3 parts of this cycle with the ***ASS*** (*Abstraction, Subdomain, Sharding*) structural diagrams.
      3. I hope that all (or at least 95% of) the possible elementary system structures in *ASS* coordinates have been described and analyzed. This means that *any* reasonable software or system architecture should map to one of the patterns reviewed or a combination thereof when projected from the *design space* onto the *ASS* coordinate system. Yes, most of the synchronous monoliths will look similar, since *ASS* turns monoliths into dumb rectangles. Nevertheless, projecting a distributed system should reveal the structure that defines many of the system’s architectural properties.

## **Summary**

Most of the architectural patterns described in Parts 3 through 5 mix layers and services to balance and reconcile various aspects of *dev* (module code) and *ops* (integration of modules) complexities (see Part 1 for the definitions).

* *Layers* have low *integration* complexity (lacking separate distributed parts) but often feature a coupled and possibly complex domain logic. They are easy to start developing and to debug, but tend to become complicated and fragile as the project grows.
* *Services* have lower *code* complexity, as each single service covers a part of the application domain, but suffer in system-wide use cases due to the fact that they are hard to coordinate. It’s a kind of flexible and evolvable solution that’s very easy to get wrong for complex or coupled domains.

It is important to find and apply a set of structures that fits the project’s needs. There are some simple rules:

*It is good* to keep a coupled logic in a single component.  
*It is bad* to spread a coupled logic over multiple components, especially if the components are asynchronous or distributed.

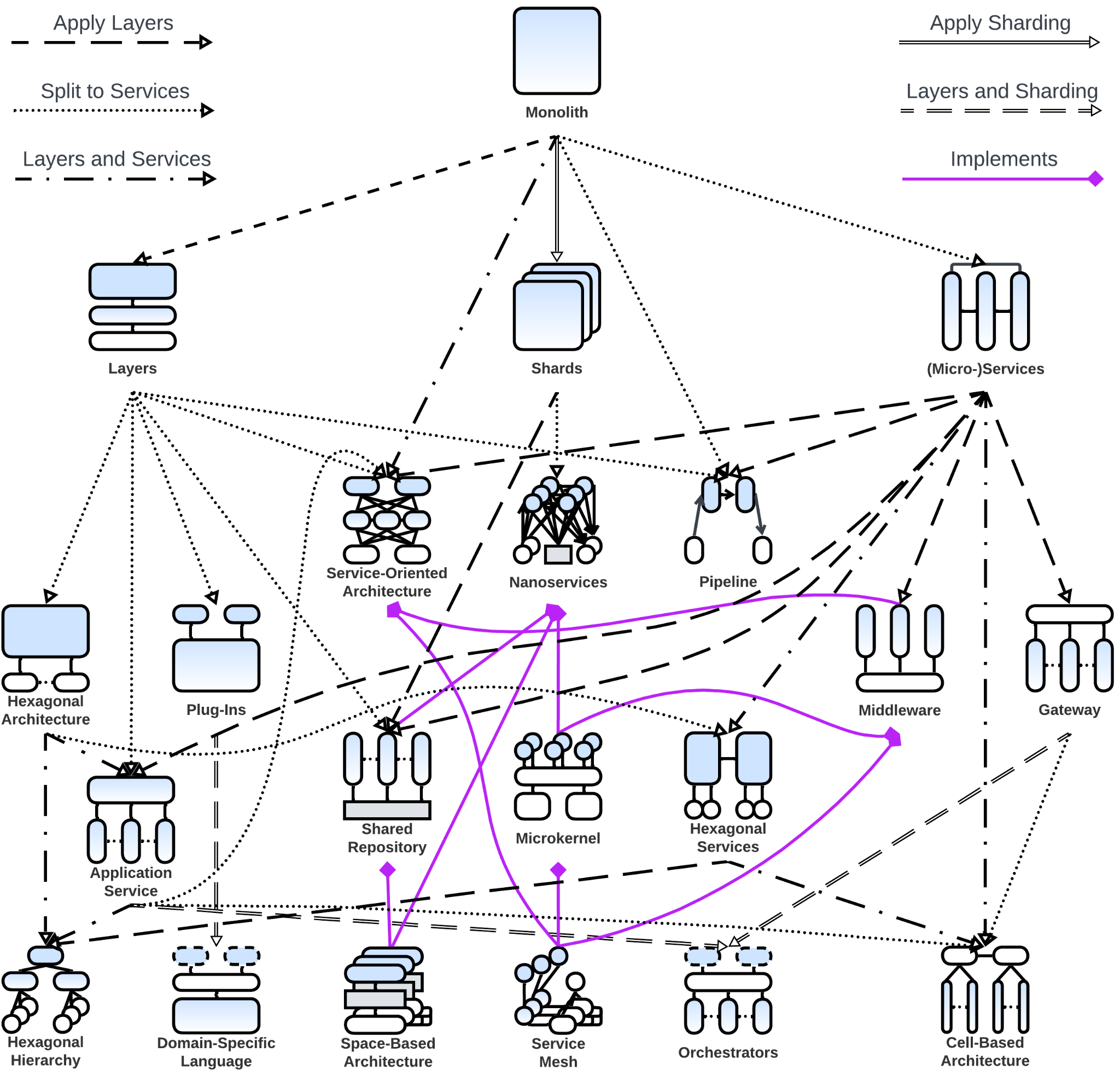
*It is good* to keep parts that vary in non-functional requirements (“-ilities” [[MP](#kix.wzzg9asvgv6o)]) apart in separate, asynchronous entities (actors or services).  
*It is bad* to try to satisfy incompatible non-functional requirements with a single component.

*It is not always possible* to follow such simple pieces of advice. Nevertheless, it is usually better to know them beforehand.  
*It is possible* to change the architecture during a project’s lifetime. However, it is easier said than done.

There is a paradox that the most important decisions must be made at the early stages of projects, precisely when the amount of information available is at its minimum. Software architecture tries to alleviate this trouble by providing means to delay the decisions involved; some patterns, e.g. *Hexagonal Architecture* (Part 4), keep the business logic protected from the implementation details, delaying the selection of third-party tools till their suitability to the project’s needs has been well tested. In other cases, it is possible to [start with a monolithic application](https://martinfowler.com/bliki/MonolithFirst.html) and delay its division into services till the project has outgrown its initial architecture. Thus, it makes sense to know in advance which architectural transitions are available.

## **The Pattern Language**

The patterns revisited in this series form a pattern system [[POSA1](#kix.y1wcg4wyxx7o)] – a classification of architectural patterns, in our case – according to their structure in *ASS* coordinates (Part 3). However, people love pattern languages [[POSA2](#kix.ksyheami8cd6)]; thus, there is a need to show how the architectural patterns are connected and how the architectures may be transformed under various forces. It should be noted that most of the transformations are reversible should forces change, e.g. eagerly escaping a monolithic hell may lead straight into a distributed transactions nightmare or an [interactions inferno](https://www.doit.com/untangling-microservices-or-balancing-complexity-in-distributed-systems/).



* 1. The diagram only plots the main transitions between architectural patterns and is too complex to show the forces behind the pattern evolutions. Thus, the reader is advised to check the “Evolution” sections in the pattern descriptions in this series to get the full explanation.

## **References**

* 1. [DDIA] Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann. O’Reilly Media, Inc. (2017).*
  2. [MP] Microservices Patterns: With Examples in Java. *Chris Richardson.* *Manning Publications (2018)*.
  3. [POSA1] Pattern-Oriented Software Architecture Volume 1: A System of Patterns. *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. John Wiley & Sons, Inc. (1996).*
  4. [POSA2] Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. *Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. John Wiley & Sons, Inc. (2000).*

* 1. [POSA5] Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. *Frank Buschmann, Kevlin Henney, Douglas C. Schmidt*. *John Wiley & Sons, Ltd. (2007).*

* 1. [SAP] Software Architecture Patterns. *Mark Richards. O’Reilly Media, Inc. (2015).*
  2. *Editor:* [Josh Kaplan](mailto:joshkaplan66@gmail.com)
  3. *Part* 1 2 3 4 **5**
  4. **Creative Commons License**
  5. This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).