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

Introduction to Software Architecture with Actors:  
Part 4 – On Systems with Models

The previous part of the cycle was dedicated to the basic ways of splitting a monolith by messaging interfaces:

1. Creating identical instances (**shards**) to scale the system under load, survive hardware failures, and enable synchronous outgoing calls (i.e. imperative programming instead of reactive programming, which is the default paradigm for asynchronous systems).

* *Precondition 1*: The data involved can be split into multiple subsets; most use cases operate on a single subset of the data.
* *Precondition 2*: There is no shared (mutable) system state that affects the use cases.

2. Dividing into **layers** to decouple a high-level business logic from low-level platform details and allow each of the layers to run independently under a distinct set of forces.

* *Precondition 1*: The amount of interaction between the layers (coupling) is much smaller than the amount of interaction inside each layer (cohesion).
* *Precondition 2*: There is no shared state that directly affects several layers.

3. Dividing into **services** to isolate subdomains, mainly for getting out of *monolithic hell* [[MP](#kix.s7pibap0d1hq)].

* *Precondition 1*: Most of the use cases don’t involve synchronized changes in several subdomains (though some use cases may chain through subdomains).
* *Precondition 2*: The subdomains don’t share state.

The preconditions [may be violated](http://www.dietmar-kuehl.de/mirror/c++-faq/big-picture.html#faq-6.16) (hinted by the word “most”) at the cost of incurring slow, complicated, and unstable hackarounds (i.e. 3-phase commit, orchestration, choreography, materialized views) that are in all aspects inferior to the direct method calls used in monoliths.

It has always been **the goal of architecture** to find a combination of applicable approaches (*patterns*) that provides strong benefits and none of whose associated drawbacks (there are no benefits without drawbacks, as everything has a price) are consequential to the project’s success. It should be noted that the importance of any given set of benefits and drawbacks depends on the nature and domain of the project in question, otherwise everyone would have long since been using the same architecture and programming language.

This article investigates multiple ways of combining **layers** and **services** to see how they work together, amplifying various benefits and mitigating some of the drawbacks.

## **The Model**

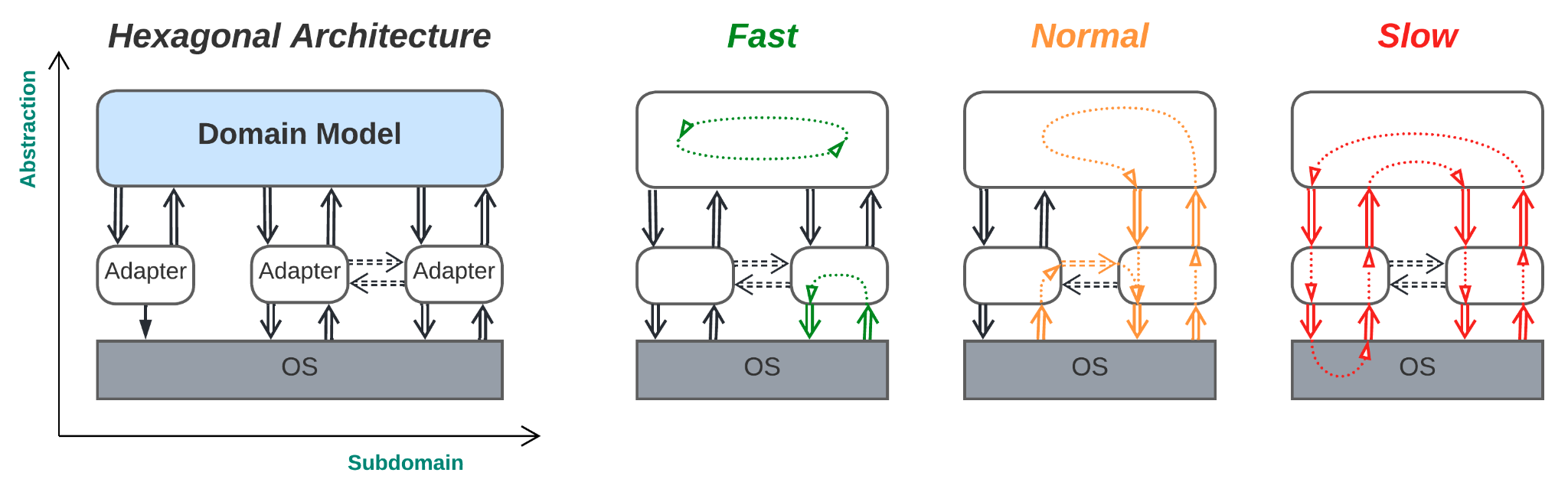
Systems with monolithic (synchronous) layers covering the whole domain are ubiquitous. Let’s call any given example of such a layer a “***model***”. The layer may be present at various levels of abstraction and may vary in thickness (the amount of logic): from the largest component that encapsulates all of the project’s business logic (e.g. in *Hexagonal Architecture*) to a thin layer providing connectivity (like in *Middleware*). A thick domain-wide layer may be called a “***domain model***”, as it contains knowledge of domain entities and, possibly, their relations. A thin layer will usually focus on a system’s components and may thus be called a “***system model***”. In a similar way, there are ***domain services*** (microservices, domain actors) that implement subdomain business logic and ***resource services*** that provide abstractions for (virtualize) system resources.

A model integrates information (control/data – see Part 2) flows over a system and sometimes links the data belonging to the subdomains involved; it unifies the components into a system and creates rules and means for the components to communicate.

## **Π-shaped systems**

* + 1. Systems may feature a monolithic model with a high-level logic that relies on the help of a lower layer of services to fulfill user requests. In such cases, the domain logic (business rules) may reside in the upper layer (the model), lower layer (the set of services) or both, creating the three following patterns, respectively*:*

### **Hexagonal Architecture** (domain model over resource services)



*(Business logic is shown in blue.)*

***Isolating the business logic from the environment****.* A monolithic layer containing the entire system’s business logic (*domain model*) runs over small services (*adapters*) that encapsulate 3rd party libraries and OS interfaces. Each adapter translates the 3rd party interface of the underlying hardware, library or remote service to the *port* (the model’s interface defined in domain terms) it serves. This structure makes the business logic module self-sufficient, as it only uses its own interfaces (*ports*) constructed in ways that are the most natural for articulating the relevant domain.

*Benefits*:

* Developing and debugging the business logic (usually about 95% of the code) is easy thanks to its monolithic structure (inherited from *Layers*), unless the project grows huge and reaches *monolithic hell* [[MP](#kix.s7pibap0d1hq)].
* The architecture can support use cases of any complexity unless they overuse periphery access (also inherited from *Layers*). Caching the most useful peripheral data in the *model* helps to further improve performance.
* The *model* (business logic) may be sharded if all the data required for calculations or decisions is included in requests (i.e. if the *model* is *stateless*) (inherited from *Layers*).
* The *model* (main logic) and each of the periphery adapters (or connectors) to external services have independent properties (e.g. real time responsiveness vs long calculations; implementation language; deployment frequency) (inherited from both *Layers* and *Services*).
* The deployment of the business logic and each of the periphery adapters is highly independent (from both *Layers* and *Services*).
* The periphery adapters run in parallel (from *Services*).
* Periphery-to-periphery communication that does not involve the main business logic layer is fast (going through direct adapter-to-adapter channels, which are optional in this architecture).
* Each of the periphery adapters, along with its underlying device or 3rd party service, is easy to replace or provide multiple implementations for, as the adapters transform the interfaces of the 3rd party components into ones suited for use with the system’s business logic (its *model*), providing protection from *vendor lock-in* and delaying vendor selection decisions till the pre-release stage of the project’s life cycle.
* The development and testing of the main business logic may start before the components it relies on are available by using *stub* adapter implementations.
* The resource-friendly autotesting of business logic using *stub* adapters is supported.
* If the model is single-threaded, bugs are reproducible by recording and replaying events from the model’s message queue.

*Drawbacks*:

* *Hexagonal Architecture* does not alleviate the project’s domain complexity (from *Layers*). The business logic is still monolithic. The use of adapter interfaces (*ports*) defined in terms that are convenient for the business logic’s domain may deal in part with the accidental complexity but cannot remedy the inherent one.
* Use cases that involve multiple peripheral devices (3rd party services) are likely to be slower and more complex with Hexagonal Architecture than with *Monolith* (though caching may often help) (from *Services*).
* The project’s start is slowed down slightly, as all the external dependencies must either be isolated by wrapping them with adapters or be mocked with stubs.

*Evolution*:

* As the project grows, *Hexagonal Architecture* is likely to fall into *monolithic hell* – a state of overwhelming code complexity that is caused by having too many people working on too large a module. The only way out is to split the *domain model* into a set of smaller *domain services*. If the domain is hierarchical (i.e. there is a single subdomain that manages or uses other subdomains), the transition to *Application Service* (see below) or more generic *Hexagonal Hierarchy* (described in Part 5) will be available. In rare cases of mostly linear data processing, the *domain model* may be turned into *Pipeline* (see Part 3). Otherwise, it should be split into *(Micro-)Services* (from Part 3). Hierarchical decomposition is usually preferable, as it decouples the business logic along both the *abstraction* and the *subdomain* dimensions, resulting in a more flexible and manageable architecture.
* A need for fine-tuned customization may be solved by building *Plug-Ins* or *Domain-Specific Language* (both described below) on top of the *domain model* of *Hexagonal Architecture*.
* When contradictory non-functional requirements arise, the business logic should be split just as it would be for dealing with the *monolithic hell* described above. However, if the project is not very complex and the domain is not strongly coupled, there is the option of going for *Nanoservices* (Part 3) and *Microkernel* (described below).
* Scalability may be achieved by either sharding the *domain model* or dispersing it into *Nanoservices* over *Microkernel*, with fault tolerance as an additional benefit.

*Summary*: *Hexagonal Architecture* brings with it many significant benefits when used for projects of a medium size and lifespan. However, small projects will be delayed by the extra effort required to code the interfaces and adapters, while huge projects will still suffer from *monolithic hell* [[MP](#kix.s7pibap0d1hq)] in the *domain model* layer.

Hexagonal architecture is derived from a monolith through the following steps:

1. *Layers* (from Part 3) are applied to split the high-level business logic from the low-level implementation details. The lower layer is called the *AntiCorruption Layer* [[DDD](#kix.ru4bhbcko3hl)] because it isolates the business logic from the details and changes in the integrated 3rd party components.
2. *Services* (also from Part 3) are applied to the lower layer so that every external dependency is wrapped with a dedicated *adapter* module, making it easy to replace or update individual 3rd party components.

The resulting architecture is both more convenient than pure *Services* (*domain actors*) and more flexible than *Layers*, combining the benefits of both patterns. The costs are:

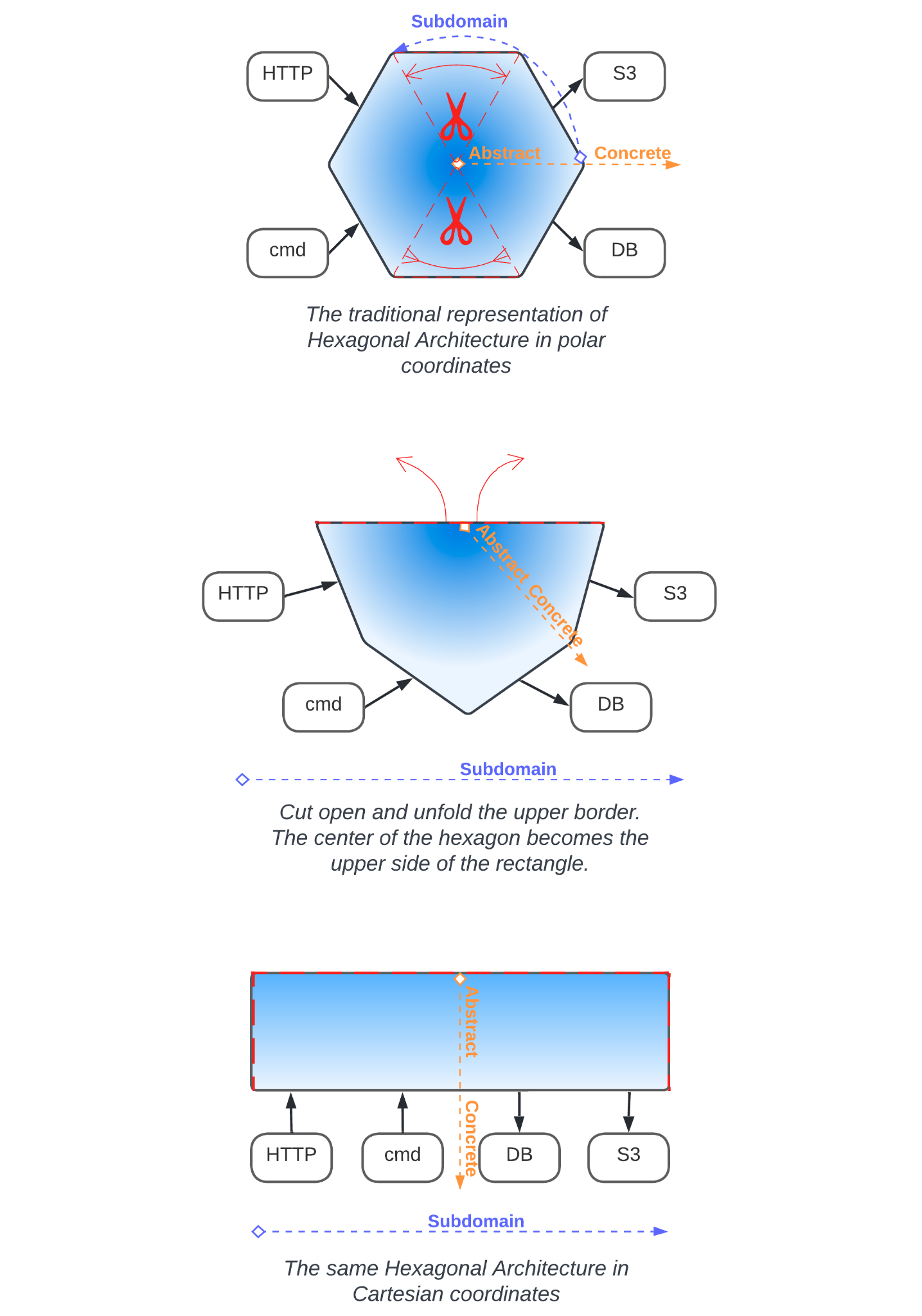
* There is no significant reduction in domain complexity (unlike that provided by vanilla *Services*)
* Use cases that are limited to a single business logic subdomain are somewhat slower than those within the original *Services* pattern.

Here, a skillful combination of two basic patterns has become ubiquitous because it collected benefits that were important to many projects without possessing any critical drawbacks. In addition, it is a prominent example of outsider architecture, something which has never been featured in famous books on patterns but has nevertheless conquered the programming world.

*Software architecture*: [Ports and Adapters](https://alistair.cockburn.us/hexagonal-architecture/), [(Re)Actor-fest](http://ithare.com/multi-coring-and-non-blocking-instead-of-multi-threading-with-a-script/3/), Half-Async/Half-Async [[POSA2](#kix.e39f5r1tdk5v)] (*Proactor* model), **Model-View-Controller** [[POSA1](#kix.mlhvarhz7mzl)] (unidirectional flow).

*System architecture*: [**Hexagonal Architecture**](https://alistair.cockburn.us/hexagonal-architecture/)**,** [Onion Architecture](https://herbertograca.com/2017/09/21/onion-architecture/), [Clean Architecture](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html).

*The diagram does not look like Hexagonal Architecture!* All the structural diagrams in this series are drawn in Cartesian *ASS* coordinates (described in Part 3), whereas typical drawings for *Hexagonal Architecture* use polar coordinates with reverse abstraction for distance and subdomain for angle:



In *control* systems, the domain model tends to contain information about the last observed state of the system and the environment to base its decisions on. For example, in robotics, a control module should be able to synchronously access (read the cached data for) both the coordinates and velocities of all the manipulators and a 3D map of its environment to allow for the real time (in-RAM, without reading from the hardware) calculations of future actions.

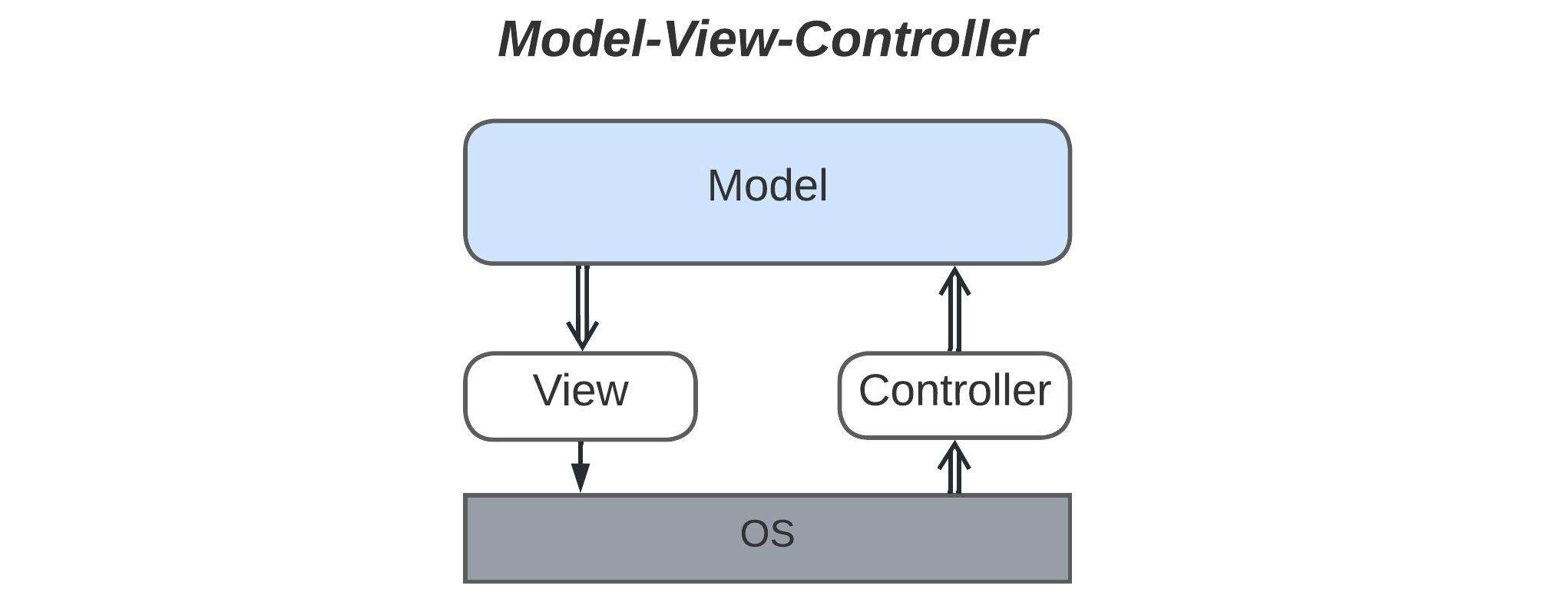
In *data processing* systems, the amount of data (the full state of the system) is usually larger than the available RAM; thus, it needs to be stored in a database wrapped with one of the adapters and be read into the model only when the data is needed to process a request. The model usually only contains the data loaded for the ongoing requests, though sometimes, it may cache frequently used objects to speed up request processing and unload the database.

Requests from the model to the adapters may be synchronous (blocking RPC), asynchronous (request/confirm or notifications), or a mixture of the two (e.g. synchronous to the database and asynchronous to the network adapters) – whichever is most convenient for the system under development based on the domain and non-functional requirements.

If both *control flow* and *data flow* are used (as in telecom), the model (called the *control plane*) is responsible for *setting up* the dataflow, while the heavy data traffic (*data plane*) is handled by the adapters (or even the underlying hardware) according to the installed rules, often in a zero-copy or DMA manner. This is where direct adapter-to-adapter channels [appear](https://hillside.net/plop/2020/papers/poltorak.pdf).

A variant:

#### **Model-View-Controller** [[POSA1](#kix.mlhvarhz7mzl)]



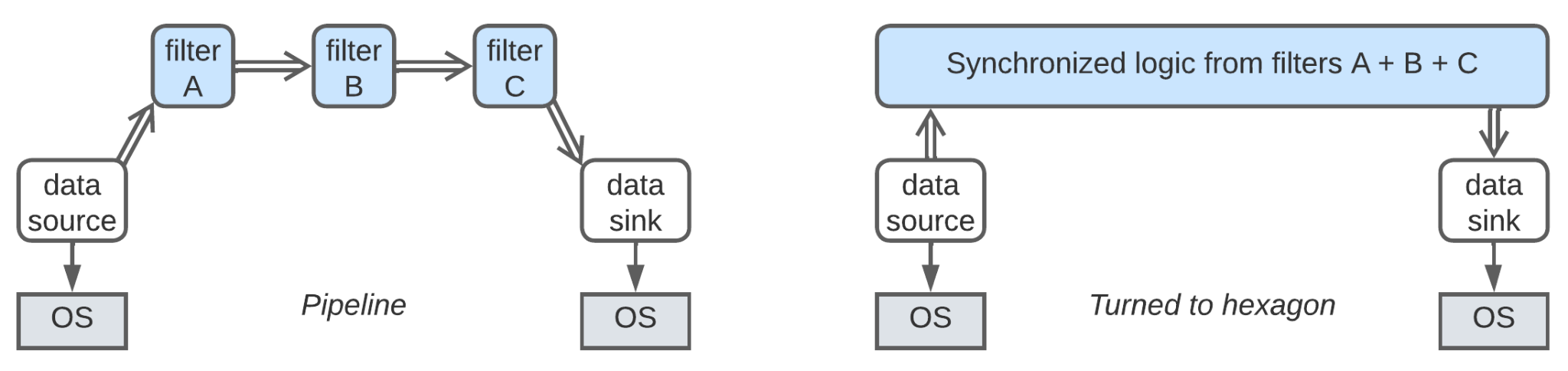
***Isolating the business logic from the UI****. B*ased on its original description wherein *view* and *controller* were adapters for a graphical interface, *MVC* may be considered a special case of *Hexagonal Architecture* with a unidirectional (*pipelined*) *control* flow.

According to [[POSA1](#kix.mlhvarhz7mzl)], *MVC* separates a *model* (logic) from a user interface in order to satisfy the following forces:

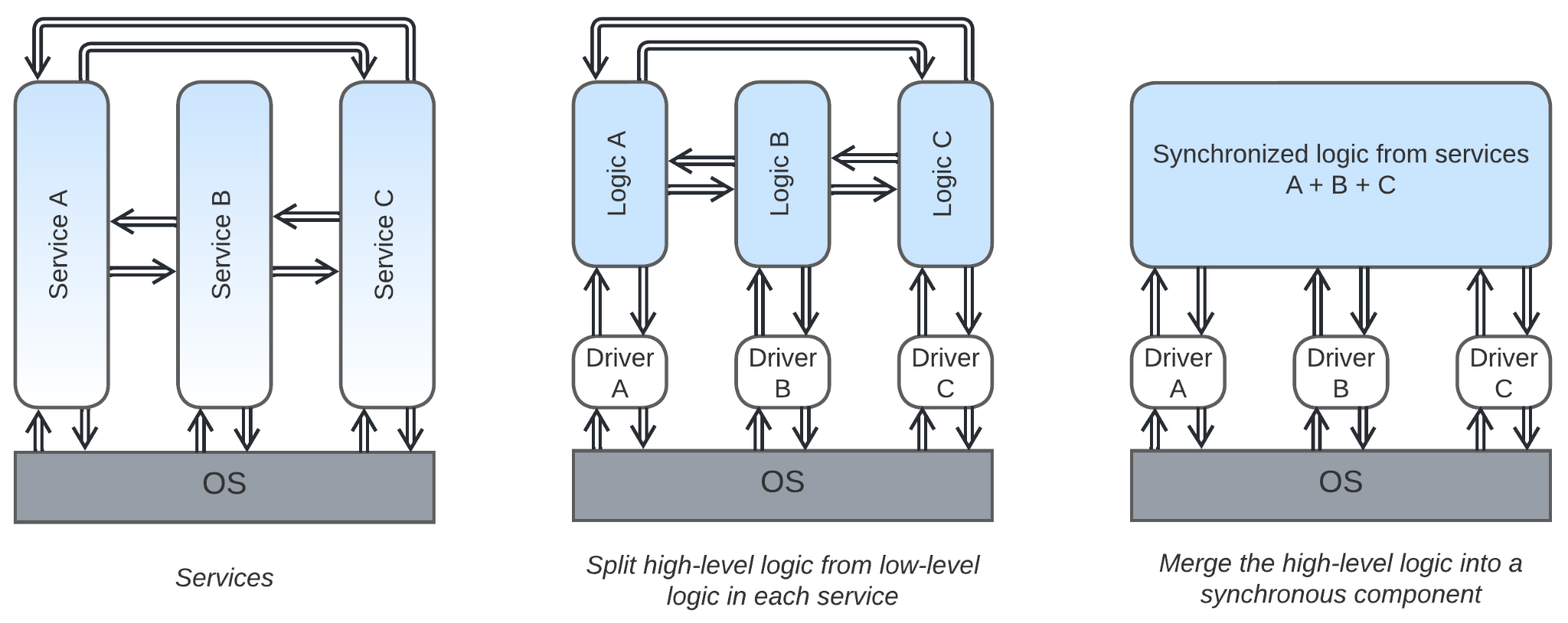
* Support for multiple *views* of the same *model*, while *Hexagonal Architecture* allows for running the same business logic (*model*) in different setups (sets of *adapters*).
* The implementation of real-time change propagation to *views*, while *Hexagonal Architecture* makes the properties (including the responsiveness) of the *model* and the *adapters* independent; thus, a slow *adapter* or *model* will not block other components.
* The option of allowing the UI to be changed without touching the main business logic, the same freedom that holds for the *adapters* in *Hexagonal Architecture*.
* The ease of porting the UI to new platforms, which is similar to the protection from *vendor lock-in* in *Hexagonal Architecture*.

Noticeably, the two old famous architectural patterns that feature nearly identical *ASS* (structural – see Part 3) diagrams *serve similar goals* and *provide similar benefits*.

The structural diagram for *MVC* also bears a resemblance to the one for *Pipeline* (Part 3); both feature unidirectional control flows and pairs of low-level components (*source* and *sink*), though in *Pipeline* architecture, the upper (business logic) layer contains multiple modules (*filters*). Thus, *Pipeline and Hexagonal Architecture are related*. If a project that was designed as a pipeline gets new requirements that make its *filters* interdependent, or if there is a need for the ultimate optimization of performance and response time, the logic-containing *filters* may be merged into a synchronous *model*, thus violating the original *Pipeline* architecture and transforming it into a *Hexagonal Architecture*. Such an experience would mean that the original understanding of the project’s domain or forces (non-functional requirements) was incorrect, but, luckily, *Pipeline* architecture is flexible enough to be reshaped at late stages of the project.

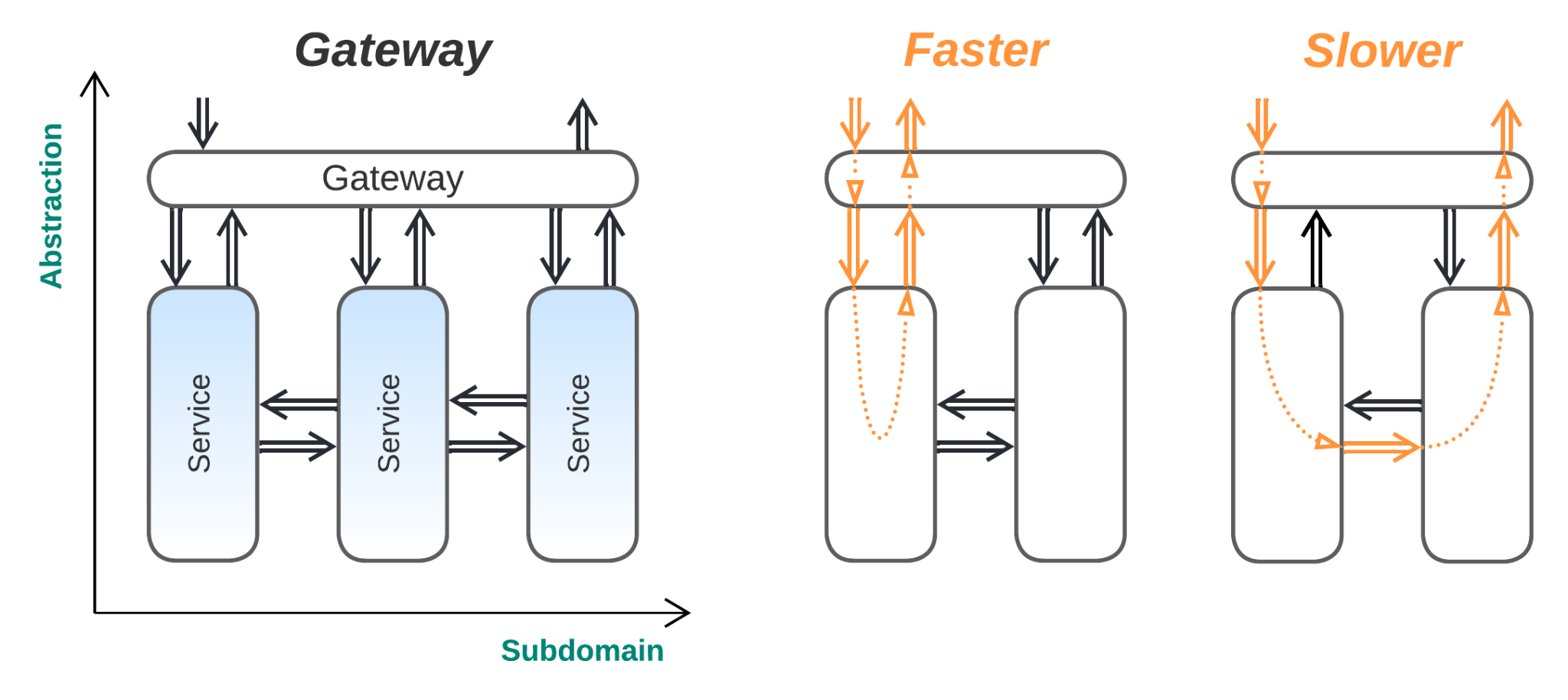


A [little more work](https://hillside.net/plop/2020/papers/poltorak.pdf) is needed to turn *domain actors* (*Services* – also from Part 3) into *Hexagonal Architecture*. First, each *service* is divided into a high-level logic and low-level implementation details layers in a way similar to the one described above for turning a monolith into *Hexagonal Architecture*. Then, the upper actors that contain the business logic belonging to the individual *services* are merged into a unified synchronous *domain model*.



With *Hexagonal Architecture*, the business logic resides in the upper layer. It is possible to reverse this setup:

### **Gateway** [[MP](#kix.s7pibap0d1hq)] (system model over domain services)



***Isolating the system from its users***. A thin system model layer is added over a set of services. The model encapsulates the services by providing a firewall and authentication functionality to the system, translating incoming requests to an internal data format, routing those requests to services of interest, and possibly aggregating responses from several services (*Mediator* [[SAP](#91zotxrfto5b)] – see the *Orchestrators* pattern below). This makes the system of services appear as a single service to the external world and reduces its attack surface.

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

* The system is encapsulated by a *gateway* that reduces the attack surface (from *Layers*).
* It is easy to support multiple client protocols, as the clients interact with a dedicated external communication module (*gateway*) that encapsulates the protocol translation and always queries the *domain* *services* using an internal protocol (from *Layers*).

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

* The *gateway* causes some performance degradation (from *Layers*).
* The *gateway* may be a single point of failure (from *Layers*).

*Evolution*:

* If the business logic becomes coupled, the *gateway* may incorporate support for *orchestrators* (see below).
* When having too many services independently developed and deployed resembles a nightmare, the services should be gathered into groups (*Cell-Based Architecture*, mentioned in Part 5), with each group encapsulated by a dedicated *gateway*.

*Summary*: A *gateway* isolates the system from external clients in a role similar to that of an *adapter* from *Hexagonal Architecture*, making it easy to support new client technologies and new types of clients without modifying the main business logic.

*Common names*: **Gateway**, Firewall.

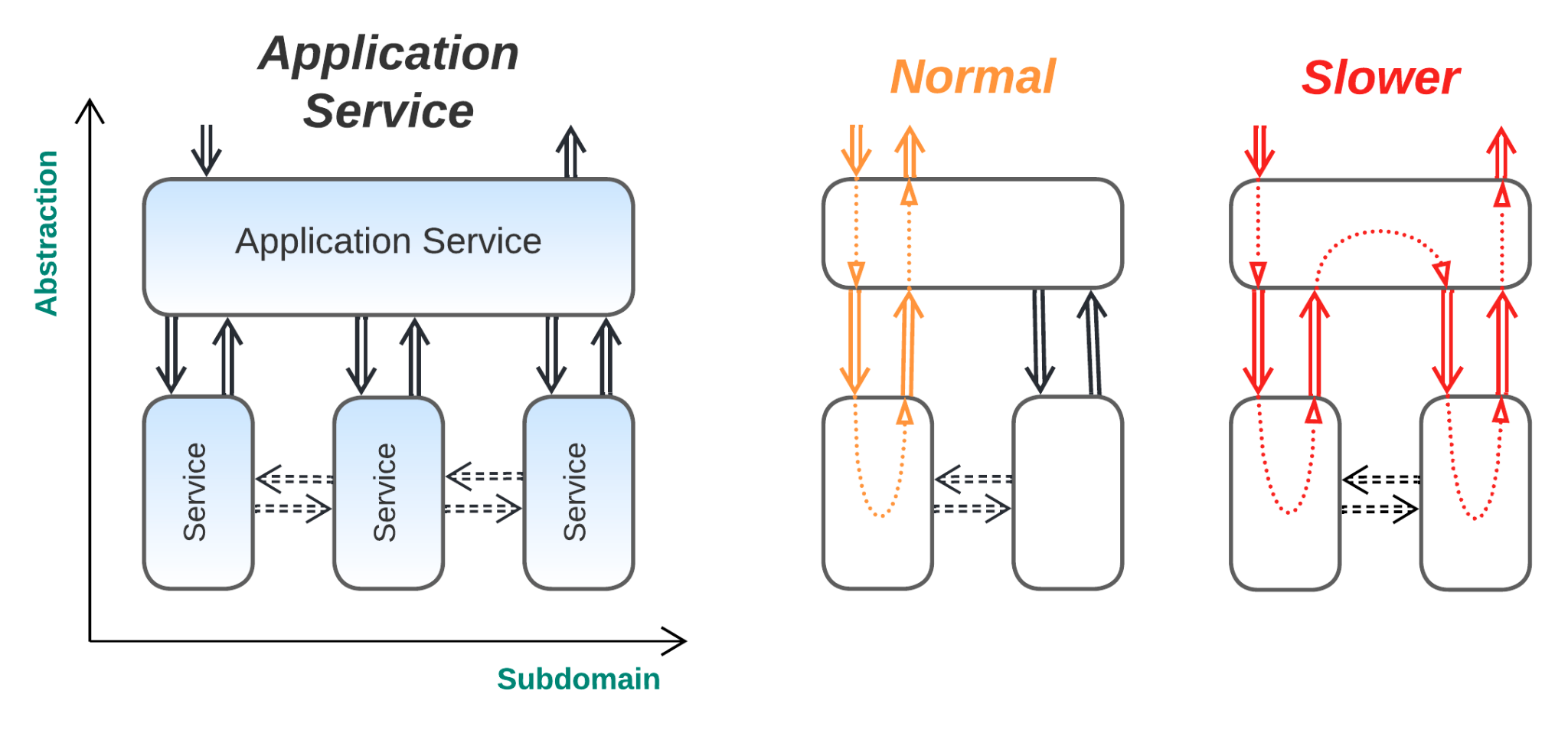
*Software architecture*: Facade [[GoF](#vhy33ai98z9c)], Proxy [[GoF](#vhy33ai98z9c), [POSA1](#kix.mlhvarhz7mzl)], Dispatcher [[POSA1](#kix.mlhvarhz7mzl)].

*System architecture*: API Gateway [[MP](#kix.s7pibap0d1hq)].

Both *Hexagonal Architecture* and *Gateway* provide an isolation of the application’s business logic from the environment. Both achieve this by adding an extra layer of indirection at the system’s boundary. They differ in the definition of the application being protected: for *Hexagonal Architecture*, the application consists of the monolithic business logic, while with *Gateway*, the whole system of services, including their databases and 3rd party components, is protected. One of the adapters in *Hexagonal Architecture* may function as a *gateway* to clients.

The only case remaining with the same structural diagram appears when the business logic is spread over all the modules:

### **Application Service** (domain model over domain services)



*(The domain services may or may not communicate directly)*

***All services are equal, but some services are more equal than others****.* There is a service (*application service*) that faces most of the customer requests and coordinates other services (*domain services*) for running system-wide tasks.

*Benefits*:

* The thorough decomposition of business logic by both subdomain and level of abstraction makes the system evolvable (from both *Layers* and *Services*).
* Coupled use cases are supported (from *Layers*).
* It is possible to scale the high-level task logic service and the subdomain services independently (from both *Layers* and *Services*).
* In many cases, domain services are not coupled (don’t rely on each other’s interfaces), unlike those in systems without the coordinator layer.

*Drawbacks*:

* Use cases are hard to code and debug (from *Services*).
* Data may need to be synchronized among the components (from *Services*).
* The system’s responsiveness and throughput deteriorate (from *Services*).

*Evolution*:

* If the business logic becomes coupled, the *application service* may incorporate support for *orchestrators* (see below).
* If the *application service* itself grows into a monolithic hell, it should either be split into *Hexagonal Hierarchy* (implementable for hierarchical domains) or become a kind of *Cell-Based Architecture*, both mentioned in Part 5.
* If a new set of use cases appear that are representable under the current domain abstraction being used by the existing *application service,* then a new implementation of *application service* may be added to cover the new use cases. This efficiently turns the *Application Service* architecture into *Service-Oriented Architecture* (see Part 5).

*Summary*: *Application Service* is a natural domain decomposition in cases where one subdomain supervises the other subdomains. It may greatly decouple business logic for huge projects at the cost of increasing operational and debugging complexity.

*Common names*: [Application Layer](https://herbertograca.com/2017/09/07/domain-driven-design/) [[DDD](#kix.ru4bhbcko3hl)].

*Software architecture*: Human-Machine Interface.

*System architecture*: [Task Service Layer](https://patterns.arcitura.com/soa-patterns/basics/soamethodology/service_layers).

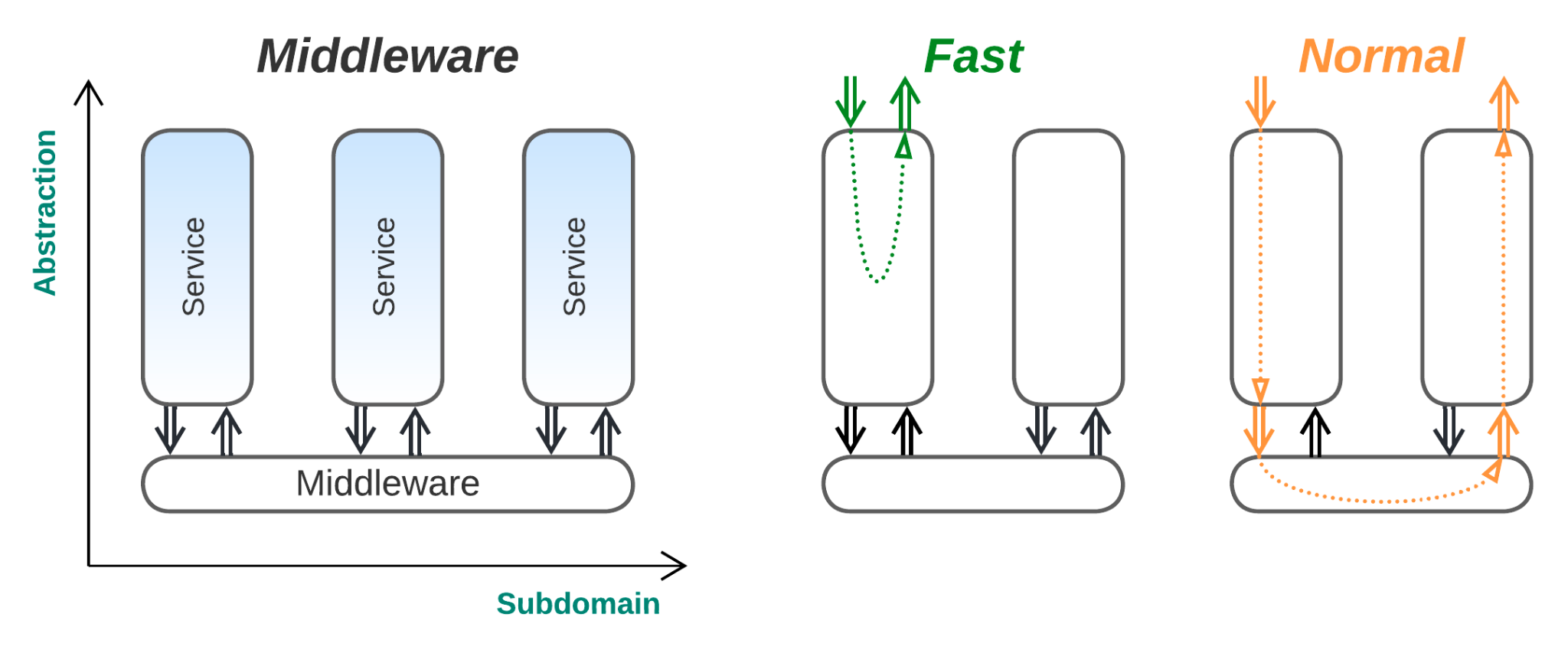
This structure is a basic *Hierarchy* (that may be built up by adding more layers of services) or an underdeveloped *SOA* (which emerges when several different *application services* are formed). Both systems are described in Part 5 and are known to be able to mitigate domain complexity, showing the possible ways of evolving a project as it grows.

If the diagram is inverted along the *abstraction* axis, it turns into:

## **U-shaped systems**

* + 1. Here, the domain services have more abstract (i.e. high-level) logic than that of the model they use, which may just consist of a service transport layer, hold shared data, or be a full domain model exposing an interface for the higher layer to build upon. Again, three patterns emerge:

### **Middleware** (domain services over a system model)



***Sharing a transport to simplify operations****. Services* communicate via a dedicated transport layer that knows about all the system components (*Broker* [[POSA1](#kix.mlhvarhz7mzl), [EIP](#kix.cu9dmhes56cm)]) and lets them address each other (*Message Bus* [[EIP](#kix.cu9dmhes56cm)]). A *middleware* may guarantee message delivery, simplifying recovery in case of failures. It may also provide message logging, which is useful for debugging and regression testing, and may sometimes manage the *services* by implementing recovery and scaling aspects.

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

* It allows a faster start to the project, as the transport layer (possibly with such *ops* functions as scaling and recovery) is usually available out of the box.
* The connectivity is less complex thanks to the presence of a uniform protocol and the system model (*Broker*).
* It simplifies failure recovery, as a *middleware* usually provides message delivery guarantees and may restart failed services.

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

* A generic *middleware* may not provide the optimal means of communication for every connection in the system.
* *Broker* topology may slow down messaging to some extent.
* The *broker* may become a single point of failure.

*Evolution*:

* *Microkernel* implementations, such as Akka or Erlang/Elixir, can be used as a *middleware* for projects that require good scalability and fine-tuned fault tolerance.
* *Service Mesh* (Part 5) can be used as a *middleware* with the ability to translate between various message formats and transports.

*Summary*: *Middleware* is very common when a relatively high number of services need to communicate – in that case, supporting communication channels between each pair of services becomes impractical. It does not add any significant architectural benefits or drawbacks to the system it serves.

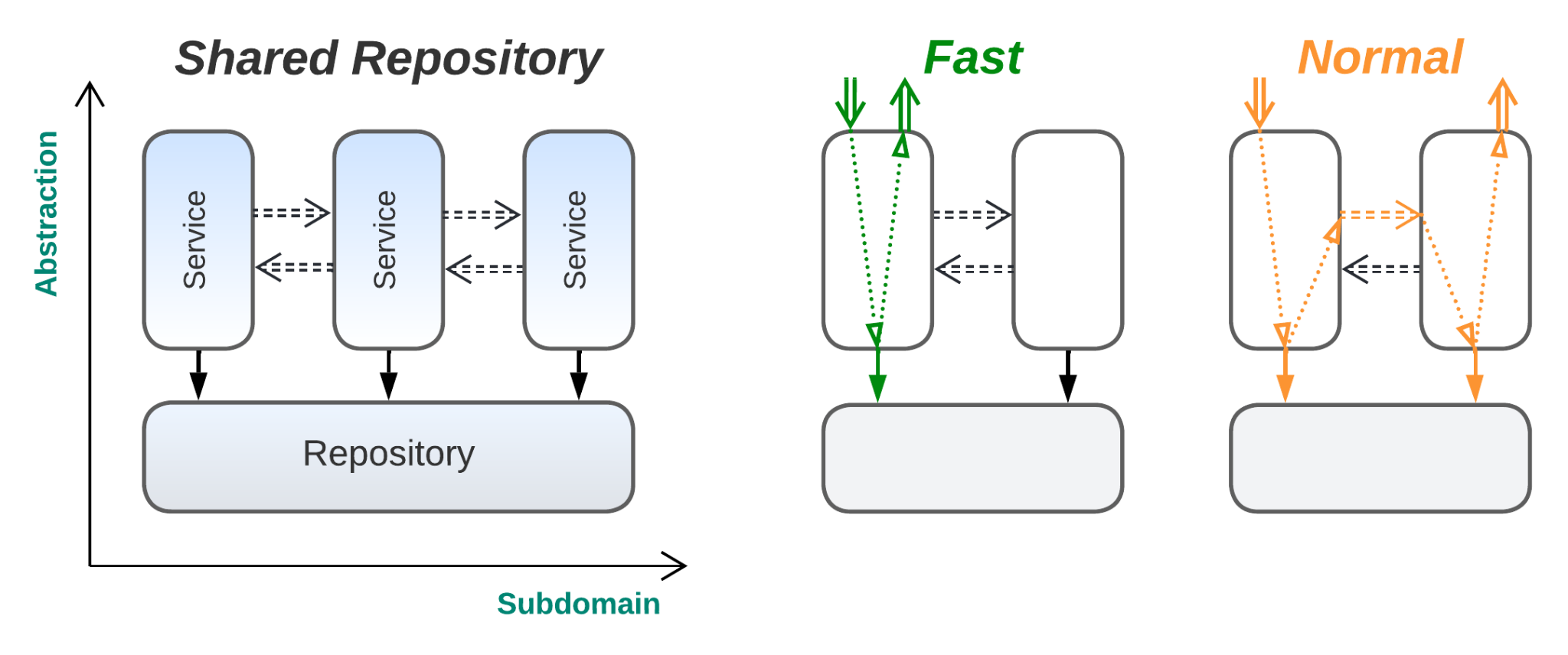
*Publish-subscribe* engines may be considered a kind of *middleware*, and they may be present even in embedded devices. A similar *middleware* option for systems of local actors is sending messages without explicitly setting destination actors; in this case, every type of message is delivered to a dedicated global subscriber (this was also briefly mentioned in Part 1).

*Software architecture*: Middleware*.*

*System architecture*: (Message) **Broker** [[POSA1](#kix.mlhvarhz7mzl), [EIP](#kix.cu9dmhes56cm), [MP](#kix.s7pibap0d1hq)], Message Bus [[EIP](#kix.cu9dmhes56cm)], Enterprise Service Bus (also manages services and translates payload formats).

The *message bus* is usually excluded from architectural diagrams, as it: is very common, does not influence business logic, and does not change the system’s properties. Aside from specialized *Middleware* frameworks*,* similar functionality is provided by decentralized *meshes* (Part 5) and resource-sharing *microkernels* (see below).

### **Shared Repository** [[POSA4](#4gtsxyq7v9rn)] (domain services over a domain *data* model)

**

*(The services may or may not communicate directly)*

***Sharing the data to simplify the development****.* Services are created with a shared *data layer* (*repository*).

*Benefits*:

* The services share data, allowing for domain-wide commits without *event sourcing* [[MP](#kix.s7pibap0d1hq)], *views* [[DDIA](#kix.37b0xh3khvut)] or *sagas* [[MP](#kix.s7pibap0d1hq)] (from *Layers*).
* The services are decoupled in deployment and some properties (from *Services*).
* The code of the services is more independent than that in vanilla *Domain Services* systems because the *data layer* mediates much of the communication.
* It allows a quicker start on the project than what would be possible with vanilla *Services* (from *Layers*).
* The *data layer* is usually available out of the box.

*Drawbacks*:

* The service implementations are still coupled via the shared *data layer* (from *Layers*).
* The *data layer* is likely to be hard to scale or shard (from *Layers*).
* The *data layer* may become a single point of failure (from *Layers*).
* The shared *data layer* implementation may not fit all the services equally well.

*Evolution*:

* As the workload and amount of stored data grows, the shared database will likely become a bottleneck. Two solutions are possible: going for vanilla *Services* (from Part 3) that don’t share any data, or relying on a distributed *data plane* of *Space-Based Architecture* (mentioned in Part 5).
* The *shared repository* may strongly couple the implementation and properties of the services that use it. Decoupling the services requires that they stop sharing data.

*Summary*: *Shared Repository* sacrifices many of the benefits of *Services* for a quick start on the project and the simplicity of its development. It holds its ground against *Monolith* (Part 2) when diverse use cases require the system to manifest contradictory properties, e.g. in basic combined OLTP + OLAP applications.

Synchronous access to a *shared repository* makes the actor system violate the actor model by sharing the state between actors. In this case, the *data layer* should take care to resolve probable deadlocks.

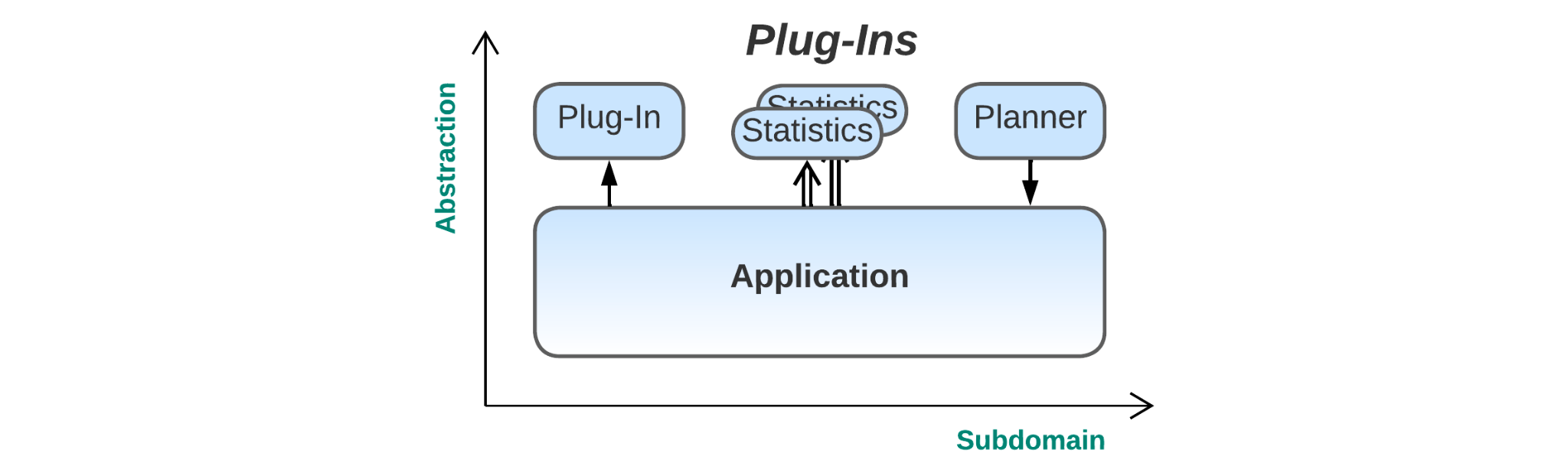
*Software architecture*: **Blackboard** [[POSA1](#kix.mlhvarhz7mzl)], Global Data.

*System architecture*: **Shared Database,** Smart UI [[DDD](#kix.ru4bhbcko3hl)], Shared Repository [[POSA4](#4gtsxyq7v9rn)].

Both *Shared Repository* and *Middleware* provide means of communication to a system of services, have similar structural diagrams, and share many benefits and drawbacks – here, once again, the correlation between a structure and its properties becomes prominent.

*Mesh* (Part 5) provides a decentralized *middleware* and sometimes also implements a *shared repository* (called *Space-Based Architecture*) with physically distributed hardware.

### **Plug-ins** [[SAP](#91zotxrfto5b)] (domain services over a domain model)

* + 1. 
    2. ***Supervising or fine-tuning the behavior***. A monolithic domain model is extended with higher-level services (*plug-ins*) that collect metadata and tune or customize the model’s behavior. The *plug-ins* may be called synchronously (i.e. looped into the model’s business logic), be notified asynchronously (e.g. collecting statistics), or query the model at will (being a controller like an AI planner, etc.).

*Benefits*:

* A very high-level logic may be written in dedicated languages or [DSL](https://en.wikipedia.org/wiki/Domain-specific_language)s (from *Layers*).
* Parts of the high-level logic are independent (in terms of development, properties, scheduling, deployment, etc.) of each other (inherited from *Services*) and of the model (inherited from *Layers*).
* The high-level logic is customizable, and different implementations may be switched in runtime or be plugged in in parallel (sharding is possible, as the *plug-ins* don’t communicate directly).
* Event replay may be applied for debugging and testing the high-level modules in isolation.
* The main business logic supports coupled use cases (from *Layers*).
* The system tends to be fast (from *Monolith*) unless *plug-ins* are abused.
* The complexity of the business logic is somewhat reduced by splitting high-level aspects into separate modules (from *Layers* and *Services*).

*Drawbacks*:

* The amount of planning and development needed to support *Plug-ins* is likely to be nontrivial.
* *Plug-ins* can only extend aspects of the model that were explicitly designed to be customizable.
* The *plug-ins* tend to be independent and may have trouble intercommunicating (from *Services*).
* Use cases that involve *plug-ins* will be slowed down (from *Layers*).
* Testability suffers greatly, as there may be many possible combinations of *plug-ins*.

*Evolution*:

* If there is a need to tune the system behavior in whole or let users write and apply custom logic, providing a *Domain-Specific Language* (see below) should be considered.

*Summary*: *Plug-ins* may benefit large projects that need extra flexibility or that have a nontrivial amount of very high-level business logic. However, the pattern often needs extensive support in the main code, which makes it quite hard to construct the system correctly on the first attempt. The approach is likely to be overkill for medium- and small-sized projects, while in huge projects, the domain model may well reach *monolithic hell* [[MP](#kix.s7pibap0d1hq)].

This is yet another case, like that of *Hexagonal Architecture*, which applies a combination of simple patterns (*Layers,* *Services* and *Sharding*) to yield benefits without inviting any critical flaws. While *Hexagonal Architecture* decouples the application’s business logic from the periphery, *Plug-ins* decouple the basic business-logic in the domain model from its high-level aspects and allow for variations in multiple aspects of the system’s behavior.

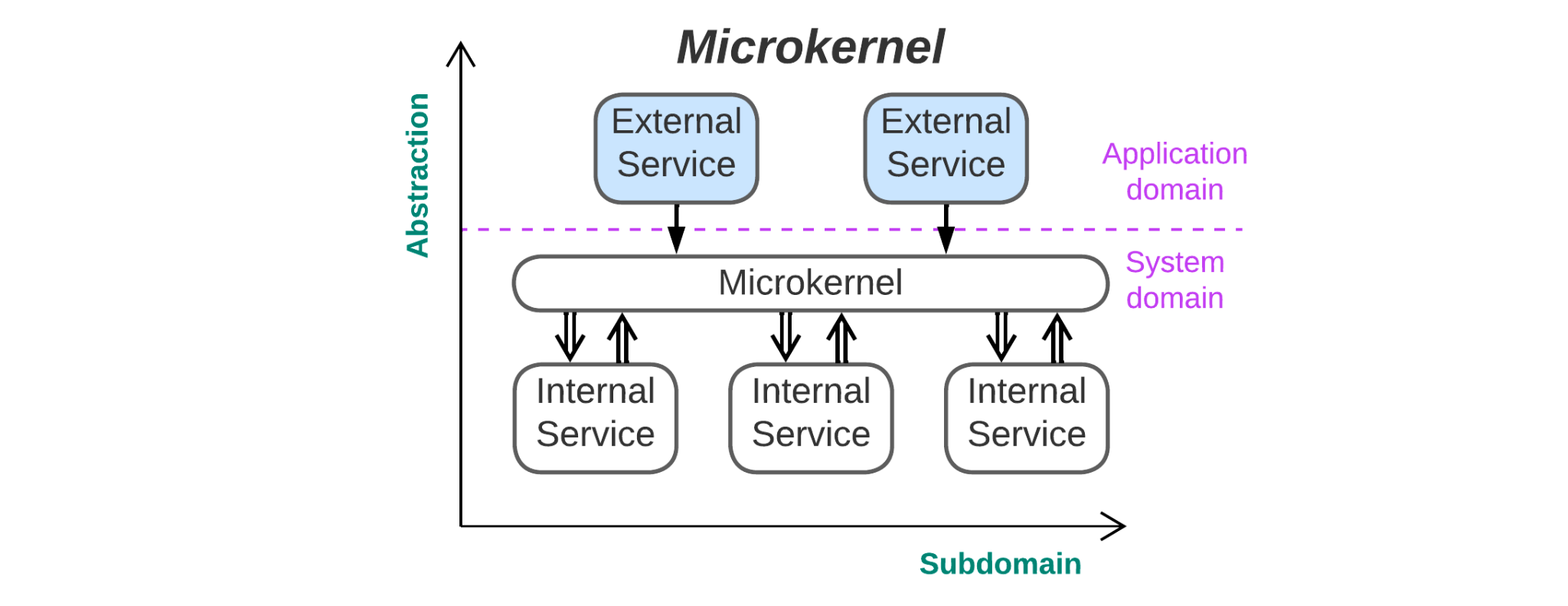
*Common names*: Metadata, Plug-ins [[SAP](#91zotxrfto5b)], Aspects, Hooks.

It is common to see *Plug-ins* applied on top of *Hexagonal Architecture*’s domain model, which results in a kind of a model-in-the-middle system and combines the properties of both architectures, but the resulting architecture does not boast any special features and thus will not be reviewed separately.

## **H-shaped systems**

* + 1. It is sometimes convenient to build a set of (often transient) *domain services* on top of a model that provides a unified or virtualized interface to the underlying system and manages the high-level services:

### **Microkernel** [[POSA1](#kix.mlhvarhz7mzl)]

* + 1. 

***Sharing the system resources among hordes of services***. A system model (*microkernel*) provides means for custom *applications* (*external services*) to communicate and access system resources owned by *drivers* (*internal services*). Usually, the *microkernel* also manages the *drivers* and *applications*, making sure that the system survives the failures of its individual components.

*By the way, here is an example of a diagram with two domains: the business domain of the applications and the resources domain of the drivers. Both are shown along the same axis*, *as only two dimensions are available for diagrams.*

*Benefits*:

* The system can often recover from the failures of its individual components (inherited from *Middleware*). This is enhanced by Erlang’s and Akka’s [supervisor hierarchies](https://ferd.ca/the-hitchhiker-s-guide-to-the-unexpected.html) if these frameworks are used.
* The system is foolproof in regard to the application’s (mis-)behavior.
* Applications can seamlessly migrate between environments (inherited from *Hexagonal Architecture*).
* Scalability is very good when distributed *Microkernel* frameworks are in use.
* It is easy to replace application dependencies, in terms of both hardware and 3rd party software (from *Hexagonal Architecture*).
* The applications that run on the same system and share its resources may be independent (from *Services*).
* The deployment cycles of *applications* and *drivers* are completely independent (from *Layers*).
* A faster start to application development is possible if the lower layers (*microkernel* and *drivers*) are initially available (from *Middleware*).

*Drawbacks*:

* The infrastructure is very complex in regard to both coding and architectural decisions that must satisfy every kind of application.
* Accessing remote resources tends to be (very) slow and complex, thus making the pattern barely viable for distributed systems unless the *microkernel* is able to move a running *application* between servers (which is supported by Akka and Erlang).
* The *microkernel* may be the single point of failure.
* *Microkernel* is often implemented as a virtual machine, slowing down the running of applications.

*Evolution*:

* A distributed microkernel can be implemented with *Service Mesh* (Part 5). However, as the implementation is very nontrivial, a 3rd party product is used in most cases.

*Summary*: *Microkernel* architecture manages the use of limited resources by untrusted applications and provides them with connectivity, often supporting large numbers of independent or interacting actors and providing the means to build supervision hierarchies for error recovery. It originates with operating systems and is mostly limited to system software (OS, virtualization, distributed FS) because of the high infrastructure development cost and performance penalty imposed on accessing distributed goods. On the other hand, yet another custom *Microkernel* implementation may become a framework for *SOA* (Part 5), which is even more heavy-weight and sluggish.

*Microkernel* may be regarded as multiple *Hexagonal Architectures* sharing a set of *adapters* via a common *middleware* or as a *Middleware-Gateway* system where the *microkernel* serves as both a *gateway* to *drivers* and a *middleware* to *applications*. A distributed *Microkernel* is related to *Mesh* (Part 5), which also provides virtualization for system resources.

*Common names*: Runtime, User Space, Applications, Coroutines.

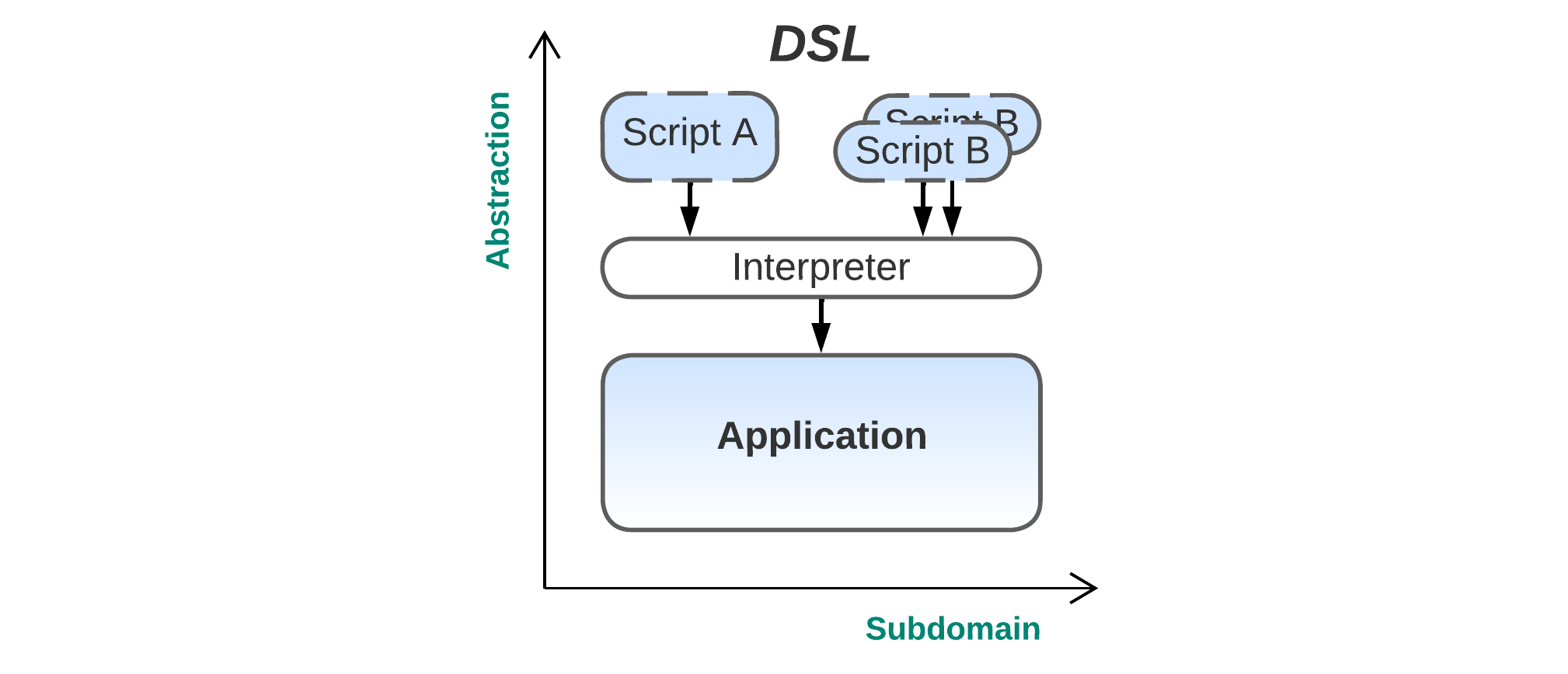
*Software architecture*: **Microkernel** [[POSA1](#kix.mlhvarhz7mzl)], Half-Sync/Half-Async [[POSA2](#kix.e39f5r1tdk5v)].

*System architecture*: **Containers**, Actors Framework.

Containerization software and Akka and Erlang runtimes are examples of distributed *Microkernel* frameworks and are used out of the box for cloud applications, some of which run [millions](https://youtu.be/bo5WL5IQAd0?t=2694) of *Nanoservices* (Part 3) with a dedicated actor instance for each user in a banking, telephony, transportation, instant messaging or online gaming network. An OS is a kind of *Microkernel*. In most cases, the *microkernel* and everything below it is skipped in drawing architectural diagrams.

*Half-Sync/Half-Async* [[POSA2](#kix.e39f5r1tdk5v)], which was first discussed in Part 2 as an implementation of *monolithic* applications then revisited in Part 3 as a *layered* pattern, appears, on closer inspection, to be *Microkernel*. Indeed, both patterns describe an OS with applications and drivers, the main difference being that *Half-Sync/Half-Async* does not discern individual drivers in the OS layer. With the coroutine implementation of *Half-Sync/Half-Async*, the coroutine engine stands for the *microkernel*, async event handlers are *internal services*, and the coroutines match *external service* components.

### **Domain-Specific Language (DSL)**

* + 1. 
    2. *(Scripts are usually transient)*
    3. ***Scripting and configuring the system***. *Plug-ins* architecture is modified by adding an intermediary layer that provides a runtime support (*interpreter*) and a kind of system model (*framework*) to the upper layer’s plugins (*scripts*), which are usually written in a high-level language. The *interpreter* will usually manage multiple instances of different *scripts*, making the whole system quite similar to *Microkernel* except for the nature of the services the lower layer provides; in *Microkernel*, the *applications* use system resources, while in *DSL*, the *scripts* manage domain objects that implement the business logic and reside in the *domain model layer*.

*Benefits* (in addition to those of *Plug-ins*):

* *Scripts* are usually very high-level and decoupled from the main codebase, thus requiring much less effort to write compared to similar functionalities in the *domain layer* (from *Plug-ins*).
* *Scripts* can seamlessly migrate between setups (from *Microkernel*).
* The system may be fool-proof against errors in *scripts* (from *Microkernel*).

*Drawbacks* (in addition to those of *Plug-ins*):

* It may be hard to design a good interface between the *scripts* and the *domain model* (from *Microkernel*).
* *Scripts* tend to be slow unless compiled, but compiling a DSL is a nontrivial task (from *Microkernel*).

*Summary*: This is a mixture of *Plug-ins* and *Microkernel* that combines most of their properties. The added *interpreter* *layer* provides a holistic way to govern the system’s behavior as a whole, as opposed to a discrete set of aspects covered by individual *plug-ins*. While *plug-ins* tend to be static (don’t change between the system’s deployments), *scripts* are often short-running tasks that set up domain rules.

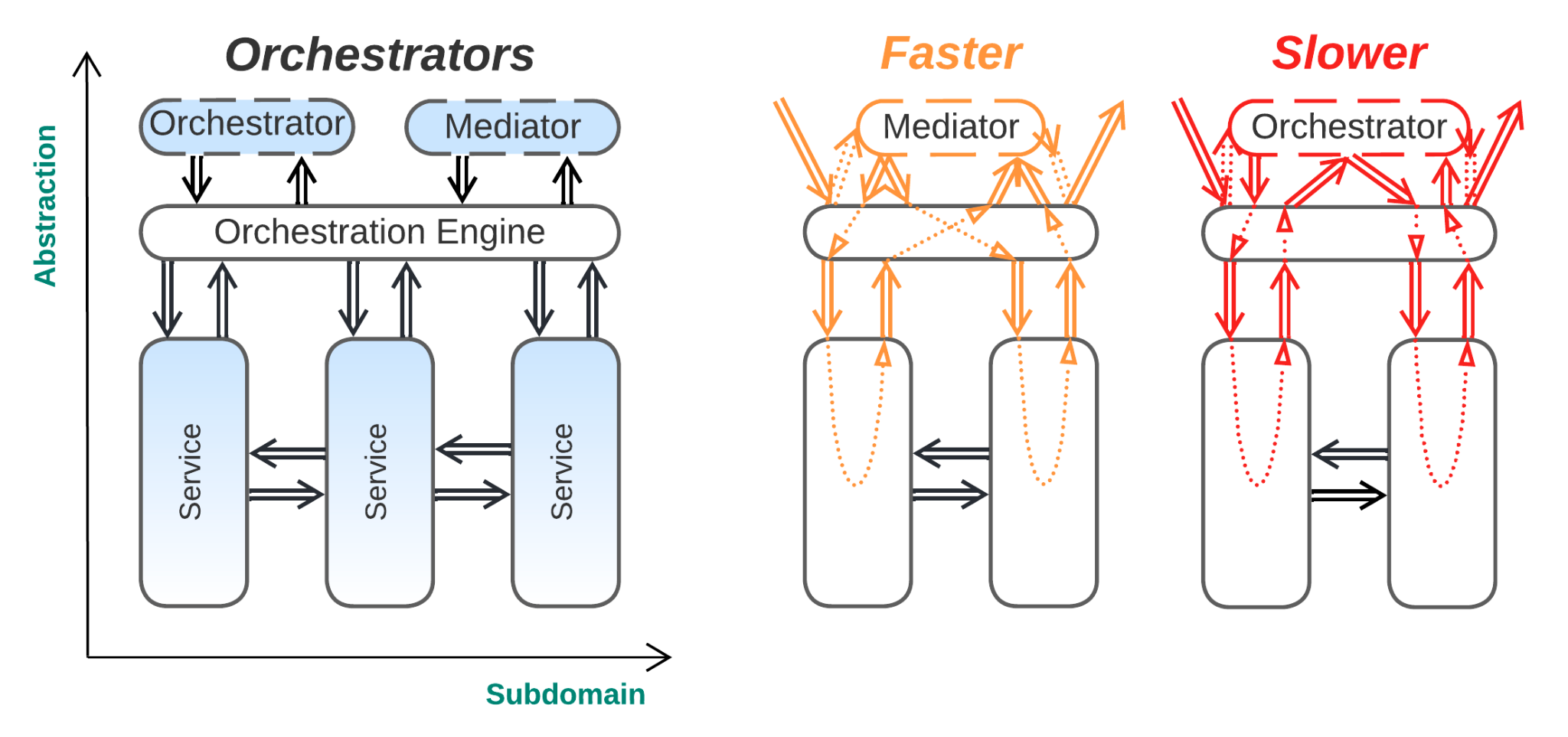
*Common names*: Domain-Specific Language (DSL), Scripts, Config File, Command Line Interface (CLI).

*Software architecture*: Interpreter [[GoF](#vhy33ai98z9c)].

Prominent examples include game scripts, Blender scripts and even SQL. A very special case of a script is a config file, which is executed once on system startup. A CLI is yet another common example.

There is a variant where the lower layer consists of domain services, with an *ASS* diagram (Part 3) similar to that of *Microkernel*. In that case, the *interpreter* is also a *gateway* for the *internal services*, while the *scripts* manage the distributed system. An example would be CLI and configuration tools for distributed systems or containerization software. Such an architecture is somewhat related to:

### **Orchestrators** [[MP](#kix.s7pibap0d1hq)]



*(Orchestrators and Mediators are transient)*

***Distributed transactions and multi-service queries****.* Multiple instances of thin domain services (*orchestrators*) are spawned by a *model layer* (a *gateway* with an orchestration engine or an *application layer*) over a set of *domain services*. Each *orchestrator* drives a multi-service use case by iterating through a sequence of steps (requests to individual services). The business logic is split between the *orchestrators* that belong to the *application layer* of [[DDD](#kix.ru4bhbcko3hl)] (and contain very high-level descriptions of the steps to be taken for each user scenario) and the underlying *services* that correspond to the *domain layer* of [[DDD](#kix.ru4bhbcko3hl)], where all of the business rules and knowledge [reside](https://herbertograca.com/2017/09/07/domain-driven-design/). Each *orchestrator* supervises a single use case running in the system, and thus, multiple instances of many types of *orchestrators* are often present simultaneously.

*Benefits*:

* The highest-level logic of use cases is easy to implement and read (from *Layers*).
* Subdomain services may be very independent, as the high-level coordination logic has been moved to a separate level.
* Domain complexity is addressed, as most of the business logic is distributed between multiple services of comparable size (from *Services*).
* The system is encapsulated with a *gateway* that reduces the attack surface (from *Gateway*).
* Multiple client types and protocols are easy to support (from *Gateway*).
* The *subdomain services* may be developed and deployed relatively independently (from *Services*).
* The *subdomain services* run in parallel and are sharded mostly independently (from *Services*).
* Use cases that don’t cross subdomain borders (i.e. are limited to a single subdomain) are very fast (from *Services*).

*Drawbacks*:

* When several *orchestrators* access the same data, it may lead to many kinds of inconsistencies or data corruption (*anomalies* caused by the *CAP theorem* [[MP](#kix.s7pibap0d1hq)]) (from *Services*, but not as severe as the corresponding drawback present there).
* Use cases that involve several subdomains are relatively slow and are hard to debug (from *Services* and *Gateway*).
* The *services* become somewhat bound in properties to allow for efficient orchestration.
* There is moderate administration complexity for *service* instances, as they must be registered with the *orchestrator engine* (from *Gateway*).
* The *orchestrator engine* (usually implemented in a *gateway* [[MP](#kix.s7pibap0d1hq)]) can often be a single point of failure (from *Gateway*).

*Evolution*:

* As the complexity of distributed use cases grows, a transition to *Application Service* or *Hexagonal Architecture* (see above) may be considered.

*Summary*: *Orchestrators* counter the main drawback of *(Micro-)Services* (described in Part 3), namely the trouble with use cases that involve several subdomains. They even allow for distributed transactions (*sagas* [[MP](#kix.s7pibap0d1hq)]). As splitting to services is the only way to counter domain complexity, *Orchestrators* are likely to emerge in huge projects. However, they are both much more irksome than synchronous method calls in a monolith (or in the model in a *Hexagonal Architecture*) and tend to somewhat limit the independence of the *services’* non-functional properties.

*Services with Orchestrators* is derived from a monolith in the following steps:

1. The monolith is split into *subdomain* *services*, escaping *monolithic hell* [[MP](#kix.s7pibap0d1hq)].
2. An extra layer (*gateway*) governing multi-service use cases and transactions is created to integrate the *services*.

This results in more control over whole-system scenarios compared to pure *Services* (*Domain Actors*) without losing the benefit of addressing domain complexity. The costs are:

* Interdomain use cases are complicated and slow (compared to *Layers*), but are at least mostly manageable.
* The system is slower and more complex than it would be with the original *(Micro-)Services* pattern.

This case contrasts with *Hexagonal Architecture*; there, the goal was to combine the benefits of the two basic patterns, whereas *Services with Orchestrators* addresses the main penalty incurred by utilizing the only basic system structure (*Services*) that is plausible for huge projects.

*Software architecture*: Coordinator [[POSA3](#kix.xmxqobij87ce)].

*System architecture*: **Orchestrator** [[MP](#kix.s7pibap0d1hq)], Event Mediator [[SAP](#91zotxrfto5b)].

Somewhat distinct types of orchestrators exist: a *saga* *orchestrator* [[MP](#kix.s7pibap0d1hq)] usually drives a distributed transaction and allows for rollbacks if one of the steps fails, while an *event mediator* [[SAP](#91zotxrfto5b)] tends to provide a unidirectional flow, with the added benefit of being able to run some or all of the involved steps in parallel. The multiple instances of orchestrators may be considered separate actors (for *sagas*) or may be embedded in a layer (*mediator* / *gateway*).

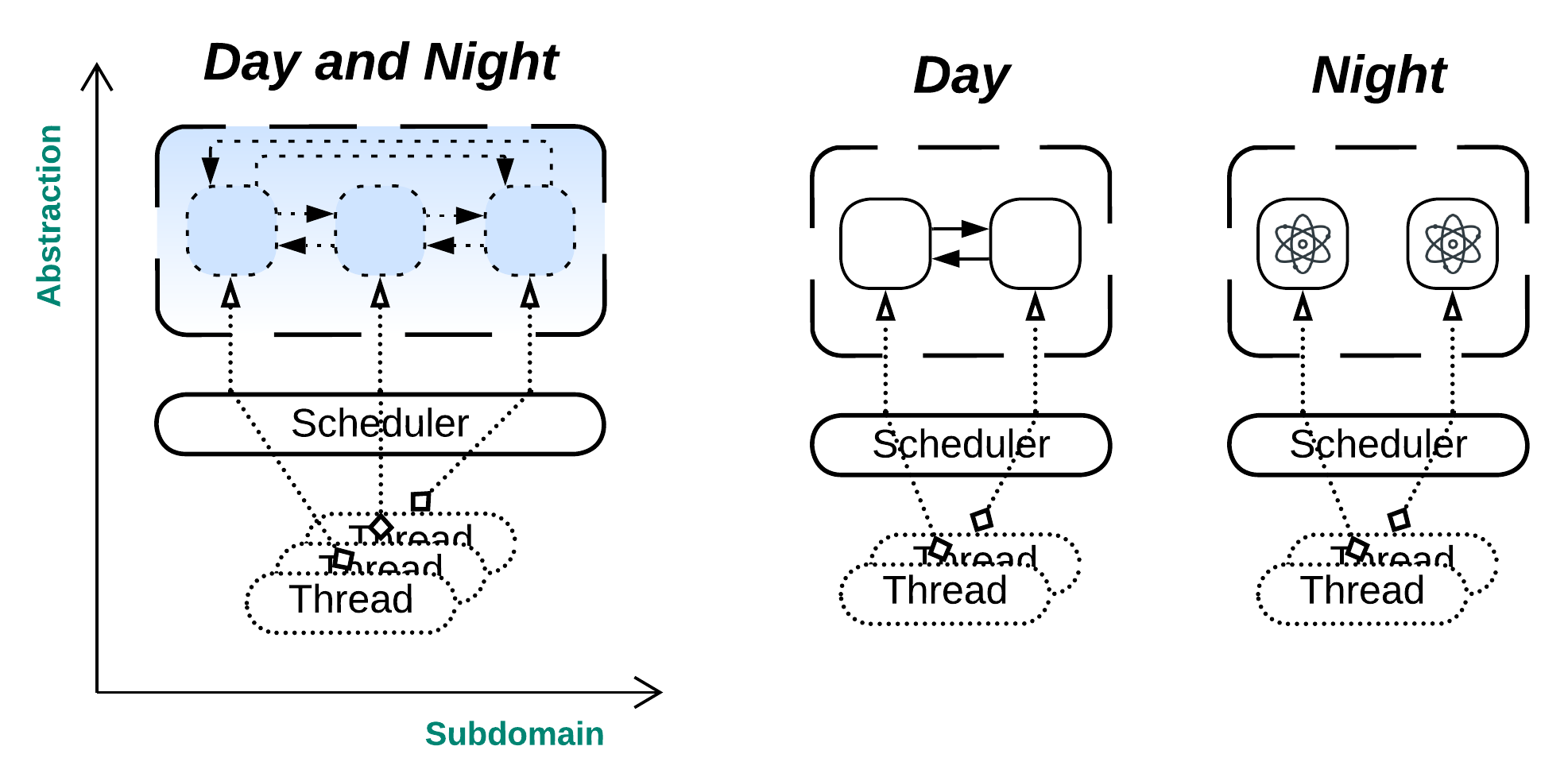
## **Miscellaneous architectures**

There are many ways to apply several of the patterns listed together, but they only seem to combine the properties of the constituents instead of creating something new:

* *Gateway* on top of *Hexagonal Architecture* (actually, the *Gateway* is yet another *adapter*)
* *Plug-ins* or *DSL* on top of *Hexagonal Architecture*
* *(Micro-)Services* with *Gateway* or *Orchestrators* and *Middleware*, *Shared Repository* or *Microkernel*
* *Hexagonal Services*, where *Hexagonal Architecture* is applied to every *service* in a system

However, there is one very special architecture unlike any other:

### [**(Re)Actor-with-Extractors**](http://ithare.com/multi-coring-and-non-blocking-instead-of-multi-threading-with-a-script/3/) **(Day and Night)**



***A lock-free shared memory interaction***. This case comes from game development and allows for the efficient use of all the available CPU cores in systems that contain large numbers of closely interacting (accessing the data of each other), self-governed (with localized decision-making) entities. There is a *scheduler* that uses a *thread pool* (a thread per CPU core) to give each entity a chance to run once per *phase*. Once all the objects are served, the *phase* is toggled. There are two kinds of *phases*:

* Read phase (*Day* or *Extract*): All the objects are read-only, thus allowing all their data and methods to be accessed without locking. The objects that are running call methods or read data from other objects they are interested in so as to read and store the data they need as input for their planning of future actions.
* Write phase (*Night* or *Process*): Every object calculates and writes its next state based on the input it has collected during the *day* phase. The objects do not interact; thus, they behave like actors, each processing a single message it posted to itself during the *day* phase.

This approach unites the shared memory (objects) and actors paradigms by leading the entire system through repeated *extract* (shared memory) and *process* (message passing) phases.

I see the next possible variants:

* The planning may be done during the *day* phase, with the planning result stored in the object’s message queue and the *night* phase amounting to the act of applying the previously calculated state updates.
* In the *day* phase, objects may post messages to each other. During the *night* phase, they will process all those messages till their queues are empty.

## **Summary:**

This part discussed the following kinds of systems that feature monolithic layers:

* Π-shaped systems, where the common layer unites and uses the underlying services.
* U-shaped systems, where the shared layer connects and supplies a shared workspace for the high-level services.
* H-shaped systems, where the layer virtualizes the underlying system for the topmost services.

Many names for mostly identical structures were mentioned. [The Rule of Three](https://wiki.c2.com/?ThingsInThrees) was observed thrice.

The next (and final) part is dedicated to fragmented architectures that contain no single module that spreads over the entire system.

## **References**

* 1. [DDD] Domain-Driven Design: Tackling Complexity in the Heart of Software. *Eric Evans. Addison-Wesley (2003).*
  2. [DDIA] Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann. O’Reilly Media, Inc. (2017).*
  3. [EIP] Enterprise Integration Patterns. *Gregor Hohpe and Bobby Woolf. Addison-Wesley (2003).*

* 1. [GoF] Design Patterns: Elements of Reusable Object-Oriented Software. *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley (1994).*
  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. [POSA3] Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. *Michael Kircher, Prashant Jain. John Wiley & Sons, Inc. (2004).*

* 1. [POSA4] Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. *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/).