

Architectural Metapatterns

*The Pattern Language
of Software Architecture*

v 1.1 (07-2025)

Denys Poltorak (author)

Lars Noodén (editor)

Licensed under Creative Commons Attribution 4.0 International

You'll find inside

- A structured collection of [architectural patterns](#) with hundreds of NoUML diagrams.
- Technology-agnostic knowledge distilled from a multitude of [sources](#).
- [Deconstruction](#) of software architecture into its basic principles.

Opentowork

I am looking for a good job. Embedded or high load C / C++. B2B from Ukraine. I can gather a team.

This book needs examples

Several readers told me that the patterns and principles should be illustrated with examples of their use in real-world systems. I cannot write them on my own because the scope of the book is much wider than my professional experience.

I am looking for both inline explanations about individual patterns (see blocks with gray background scattered throughout this book) and for one or two introductory case studies that will detail internal workings and evolution of complex real-world software to show how patterns are used in practice and promote the book to the [duplex league](#).

Please consider sharing your experience as a co-author of a future version of this book.

Short table of contents

[About this book](#)

[Metapatterns](#)

[Modules and complexity](#)

[Forces, asynchronicity and distribution](#)

[Four kinds of software](#)

[Arranging communication](#)

[Monolith](#)

[Shards](#)

[Layers](#)

[Services](#)

[Pipeline](#)

[Middleware](#)

[Shared Repository](#)

[Proxy](#)

[Orchestrator](#)

[Combined Component](#)

[Layered Services](#)

[Polyglot Persistence](#)

[Backends for Frontends \(BFF\)](#)

[Service-Oriented Architecture \(SOA\)](#)

[Hierarchy](#)

[Plugins](#)

[Hexagonal Architecture](#)

[Microkernel](#)

[Mesh](#)

[Comparison of architectural patterns](#)

[Ambiguous patterns](#)

[Architecture and product life cycle](#)

[Real-world inspirations for architectural patterns](#)

[The heart of software architecture](#)

[Appendix A. Acknowledgements.](#)

[Appendix B. Books referenced.](#)

[Appendix C. Copyright.](#)

[Appendix D. Disclaimer.](#)

[Appendix E. Evolutions.](#)

[Appendix F. Format of a metapattern.](#)

[Appendix G. Glossary.](#)

[Appendix H. History of changes.](#)

[Appendix I. Index of patterns.](#)

About this book

When I was learning programming, there was [*Gang of Four*](#). The book promised to teach software design, and it did to an extent with the case study provided. However, the patterns it described were merely random tools which had little in common. After several years, having reinvented [*Hexagonal Architecture*](#) along the way, I learned about [*Pattern-Oriented Software Architecture*](#). The series had many more intriguing patterns, and promised to provide a *system of patterns* or a *pattern language*, but failed to build an intuitive whole. Then there were specialized books with [*Domain-Driven Design*](#) and [*Microservices*](#) patterns. There was the [*Software Architecture Patterns*](#) primer by Mark Richards. Its simplicity felt great, but it had only 5 architectural styles, while his next book, *Fundamentals of Software Architecture*, dived too deeply into practical details and examples to be easily grasped.

Now, having leisure thanks to the war, burnout, unemployment and depression I have had a chance to collect architectural patterns from multiple sources and build a taxonomy of architectures. My goal was to write the very book I lacked in those early years: a shallow but intuitive overview of all the software and system architectures as used in practice, their properties and relations. I hope that it will be of some help both to novice programmers as a kind of a primer on the principles of high-level software design and to adept architects by reminding them of the big picture outside of their areas of expertise.

The book is mostly technology-agnostic. It does not answer practical questions like “Which database should I use?” Instead it inclines towards the understanding of “When should I use a shared database?” Any specific technologies ~~are easy to google~~ can be found over the Internet somewhere in the Noosphere.

This book started as a rather small project to prove that patterns can be intuitively classified (*These nightmarish creatures can be felled! They can be beaten!*) but grew into a multifaceted compendium of a hundred or so architectures and architectural patterns. It is grounded in the idea that software and system architecture evolves naturally, as opposed to being scientifically planned. Thus, the architectures may exhibit [*fractal features*](#), just like those in biology – merely because the [*set of guidelines and forces*](#) remains the same for most systems that range from low-end embedded devices to world-wide financial networks. Moreover, in some cases we can see the same patterns applied to hardware design.

The idea of unifying software and system architecture is heretical. I am well aware of that. Still, the industry is in the early stage of alchemy these days: the same things are sold under multitudes of names, being remarkedeted or reinvented every decade. If this book manages to provide rules of thumb, similar to those of biology (a bat is a mammal, thus it should run on all four, while ostriches, as birds, must fly to Europe each spring), I will be happy with that. *Science makes progress funeral by funeral.*

The latest version of the book is available for free on [GitHub](#) and [LeanPub](#). As there is no one who has practiced all the known architectures, it will be full of mistakes. I rely on your goodwill to correct them and improve the text. Critical reviews are warmly welcome: please write an [email](#) or contact me [on LinkedIn](#).

Structure of the book

The [first chapter](#) explains the main idea which makes this book different from others. The following chapters in the [first part](#) touch on several general topics that are referenced throughout the book.

The next [four parts](#) iterate over *metapatterns* (clusters of closely related architectural patterns), starting with the simplest one, namely [Monolith](#), then heading towards more complex systems that may be derived from *Monolith* by repeatedly dissecting it with interfaces. Each chapter describes a group of related patterns that share benefits and drawbacks, adds in a few references to books and websites, and summarizes the ways the patterns can be transformed into other architectures. The format of these chapters is described in [Appendix F](#).

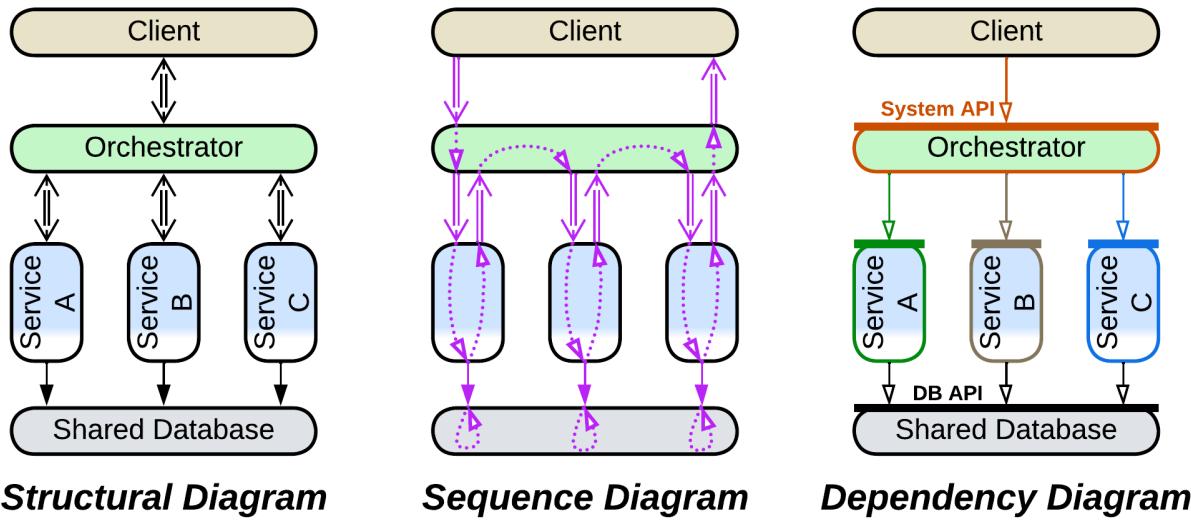
The [sixth part](#) of the book is analytics – the fruits of the pattern classification from the earlier parts.

Finally, there are appendices. [Appendix B](#) is the list of the books referenced, [Appendix E](#) contains detailed evolutions of patterns and [Appendix I](#) is the index of the patterns found in the book.

Diagrams

This book makes heavy use of diagrams – to the extent that it can be treated as a kind of visual novel. As it is mostly made of patterns, and *each pattern is an island*, it must not be read sequentially – instead, the reader is advised to use the plentiful cross-links to open whatever (if any) content found to be intriguing and check the corresponding diagram. If it gets your attention, you may read the text below it. If you like the text, you may scroll up or down to see if there are more funny diagrams nearby.

The diagrams are *NoUML* and most of them belong to one of the following kinds:



Structural Diagram

Sequence Diagram

Dependency Diagram

Please refer to the [following chapter](#) for the legend and the system of coordinates.

Notation

- Pattern names are given in [*Title Case Italics*](#) and usually link to the pattern's definition.
- The first mention of a term or a name of a pattern component is *italicized*.
- *Quotes and puns are in full italics*.

- Book references are [BRACKETED] and link to the list of the books in Appendix B.
- Supplementary explanations are grayed-out.

Many patterns match terms of the common language – indeed, as a pattern is a generalization of human experience, the more widespread a notion, the faster it is turned into a pattern. Such general-use terms, e.g. layers, services or pipeline, are usually not indicated in any way to preserve the overall readability.

The architectural religions

There are several schools of software architecture:

1. The believers in [SOLID](#).
2. The followers of [eight qualities](#), [five views](#) and [as-many-as-one-gets certifications](#).
3. The aspirants to the nameless way of [patterns](#).

In my opinion:

1. SOLID is a silver bullet that tends to produce a [DDD-layered kind of Hexagonal Architecture](#). It lacks the agility of pluralism found with evolutionary ecosystems.
2. Architectural frameworks are overcomplicated thus hard to understand and inflexible.
3. Patterns are like a kind of toolbox, the one which a mechanic is often seen carrying around. A skilled craftsman knows best uses of his tools, and can invent new instruments if something is missing in the standard toolset. However, the toolset's size should be limited for the tools to be familiar to the practitioner and easily carried around.

It is likely that those approaches are best used with systems of different sizes: SOLID is aimed at stand-alone application design while the heavy frameworks and certifications suit distributed enterprise architectures. In such a worldview patterns span everything in between the two extremes.

Patterns of software architecture are abstract just like [Plato's Ideas or Forms](#) in philosophy or classes in object-oriented programming. There is only one instance of each given pattern, which is a general idea or a very high-level blueprint for every implementation of the pattern ever seen in the code.

What's wrong with patterns

Too much information is no information or, as they say, *what is not remembered never existed*. There are literally thousands of patterns described for software and system architectures. Nobody knows them all and nobody cares to know them all (if you say you do, you must have already read [the Pattern Languages of Programs archives](#). Have you? Neither have I). Hundreds of patterns are generated yearly in just the conferences alone, not to mention the books and software engineering websites. Old patterns get [rebranded or forgotten and reinvented](#). This is especially true for the discrepancy between the pattern names in software architecture and system architecture. The new *N-tier* is just good old *Layers* under the hood, isn't it?

This undermines the original ideas which brought in the patterns hype:

1. *Patterns as a ubiquitous language.* Nowadays similar, if not identical, patterns bear different names, and some of them are too obscure to be ever heard of (see [the PLoP archives](#)).
2. *Patterns as a vessel for knowledge transfer.* If an old pattern is reinvented or plagiarized, most of the old knowledge is lost. There is no continuity of experience.
3. *Pattern language as the ultimate architect's tool.* As patterns are re-invented, so are pattern languages. At best, we have domain-specific or architecture-limited ([DDD](#), [Microservices](#)) systems of patterns. There is no *single unified vision* which pattern enthusiasts of old promised to provide.

Have we been fooled?

TLDR

Compare [Firewall](#) and [Response Cache](#). Both represent a system to its users and implement generic aspects of the system's behavior. Both are [Proxies](#).

Take [Saga Execution Component](#) and [API Composer](#). Both are high-level services that make a series of calls into an underlying system – they *orchestrate* it. Both are [Orchestrators](#).

It's that simple and stupid. We can classify architectural patterns.

Metapatterns

Is there a way to bring the patterns into order? They are way too many, some obscure, others overly specialized.

We can try. On a subset. And the subset should be:

- *Important* enough to matter for the majority of programmers.
- *Small* enough to fit in one's memory or in a book.
- *Complete* enough to assure that we don't miss anything crucial.

Is there such a set? I believe so.

Architectural patterns

[POSA1] defines three categories of patterns:

- *Architectural patterns* which deal with the overall structure of a system and functions of its components.
- *Design patterns* which describe relations between objects.
- *Idioms* which provide abstractions on top of a given programming language.

Architectural patterns are important by [definition](#) (*Architecture is about the important stuff. Whatever that is*). Point 1 (*importance*) – checked.

Any given system has an internal structure. When its developers talk about *architectural style* [POSA1] or draw structural diagrams that usually boils down to a composition of two or three well-known architectural patterns. Choosing architectural patterns as the subject of our study lets us feed on a large body of books and articles that describe similar designs over and over again. Moreover, as soon as a system no longer follows the latest fashions, it is widely advertised as a novelty (or its designers are labeled as old-fashioned and shortsighted), thus we may expect to have heard of nearly all of the architectures which are used in practice. Point 3 (*completeness*) – we have more than enough examples to analyze.

To organize a set of patterns we rely on the concept of

Design space

Design space [POSA1, POSA5] is a model that allocates a dimension for each choice made while architecting the system. Thus it contains all the possible ways for a system to be designed. The only trouble – it is multidimensional, maybe infinite, and the dimensions will differ from system to system.

There is a workaround – we can use a projection from the design space into a 2- or 3-dimensional space which we are more comfortable with. However, projection causes a loss of information. Counterintuitively, that is good for us – similar architectures that differ in small details become identical as soon as the dimensions they differ in disappear. If we could only find 2 or 3 most important dimensions that apply equally well to each pattern in the set that we want to research, that is architectural patterns, which cover all the known system designs.

Structure determines architecture

Systems tend to have an internal structure. Those that don't are derogatively called [*Big Balls of Mud*](#) for their peculiar properties. Structure is all about components, their roles and interactions. Many architectural styles, for example, [*Layers*](#) and [*Pipeline*](#), are named after their structures, while others, like [*Event-Driven Architecture*](#), highlight some of its aspects, hinting that it is the structure which determines principal properties of a system.

I am not the first person to reach such a conclusion. *Metapatterns* – clusters of patterns of similar structure – were [defined](#) shortly after the first collections of design patterns had appeared but they never made a lasting impact on software engineering. I believe that the approach was applied prematurely to analyze the [\[GoF\]](#) patterns, which make quite a random and incomplete subset of design patterns, resulting in an overgeneralization. I intend to plot structures of all the architectural patterns I encounter, group patterns of identical structure together into metapatterns, draw relations between the metapatterns, and maybe show how a system's structure determines its properties. Quite an ambitious plan for a short book, isn't it?

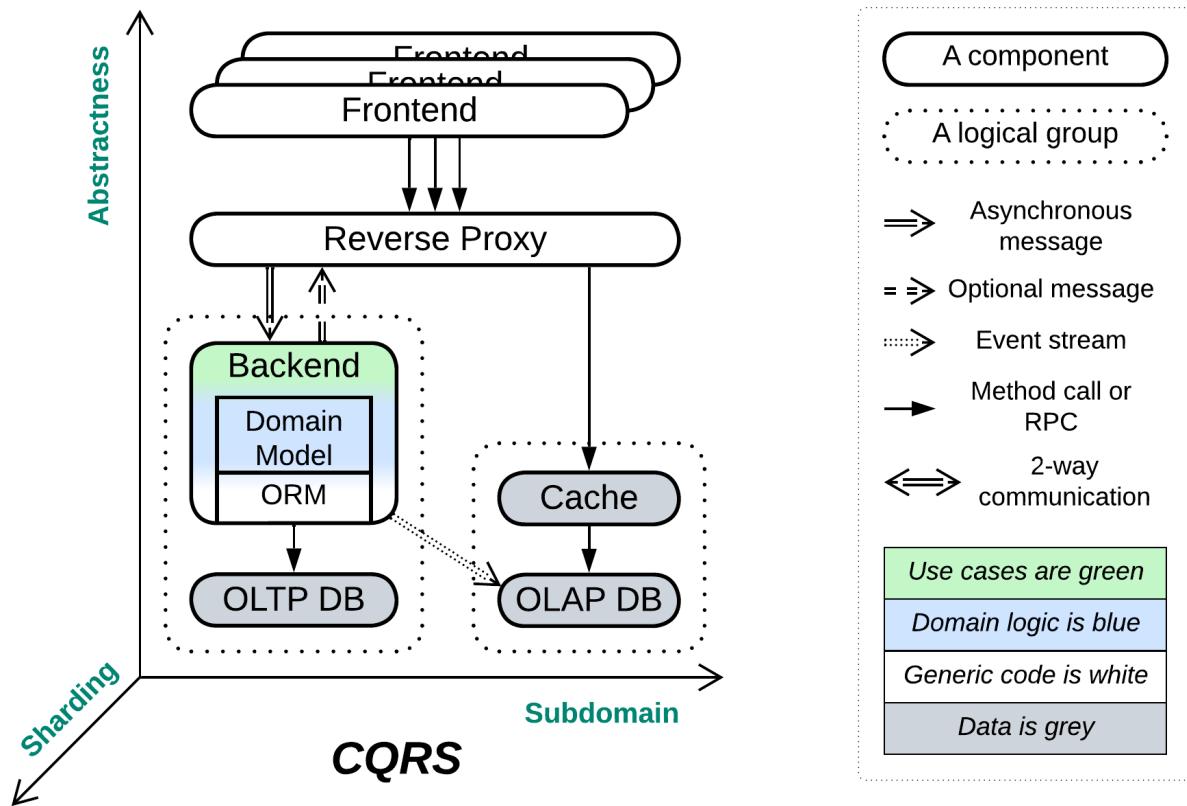
Our set of architectural patterns is still not known to be complete, is not small and, moreover, the way structural diagrams are drawn differs from source to source – we cannot compare them unless we make up a universal system of coordinates.

The system of coordinates

Inventing a generic coordinate system to fit any pattern's representation, from [*Iterator*](#) [\[GoF\]](#) to [*Half-Sync/Half-Async*](#) [\[POSA2\]](#), may be too hard, but we surely can find something for architectural patterns, as all of them share the scope, namely the system as a whole. Which dimensions an implementation of a system would usually be plotted along?

1. *Abstractness* – there are high-level use cases and low-level details. A single highly abstract operation unrolls into many lower-level ones: Python scripts run on top of a C runtime and assembly drivers; orchestrators call API methods of services, which themselves run SQL queries towards their databases which are full of low-level computations and disk operations.
2. *Subdomain* – any complex system manages multiple subdomains. An OS needs to deal with a variety of peripheral devices and protocols: a video card driver has very little resemblance to an HDD driver or to the TCP/IP stack. An enterprise has multiple departments, each operating a software that fits its needs.
3. *Sharding* – if several instances of a module are deployed, and that fact is an integral part of the architecture, we should represent the multiple instances on our structural diagram.

We'll draw the abstractness axis vertically with higher-level modules positioned towards the upper side of the diagram, the subdomain axis horizontally, and sharding diagonally. Here is an (arbitrary) example of such a diagram:



(A structural diagram for [CQRS](#), adapted from [Udi Dahan's article](#), to introduce the notation)

Map and reduce

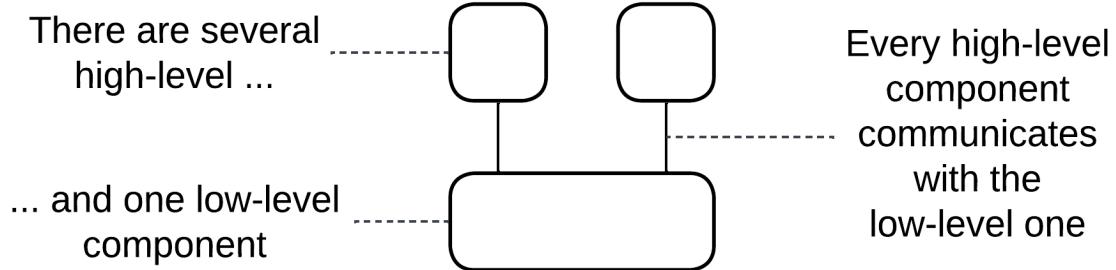
Now that we have the generic coordinates which seem to fit any architectural pattern, we can start mapping our set of architectural patterns into that coordinate system – the process of reducing the multidimensional design space to the few dimensions of structural diagrams which we were looking for. Then, after filtering out minor details, our hundred or so of published patterns should yield a score of clusters of geometrically equivalent diagrams – just because there are very few simple systems that one can draw on a plane before repeating oneself. Each of the clusters will represent an *architectural metapattern* – a generalization of architectural patterns of similar structure and function.

Let's return for a second to our requirements for classifying a set of patterns. The importance (point 1) of architectural patterns was proved before. The reasonable size of the resulting classification (point 2) is granted by the existence of only a few simple 2D or 3D shapes (metapatterns). The completeness of the analysis (point 3) comes from, on one hand, the geometrical approach which makes any blank spaces (possible geometries with no known patterns) obvious, and on the other – from the large sample of architectural patterns which we are classifying.

Godspeed!

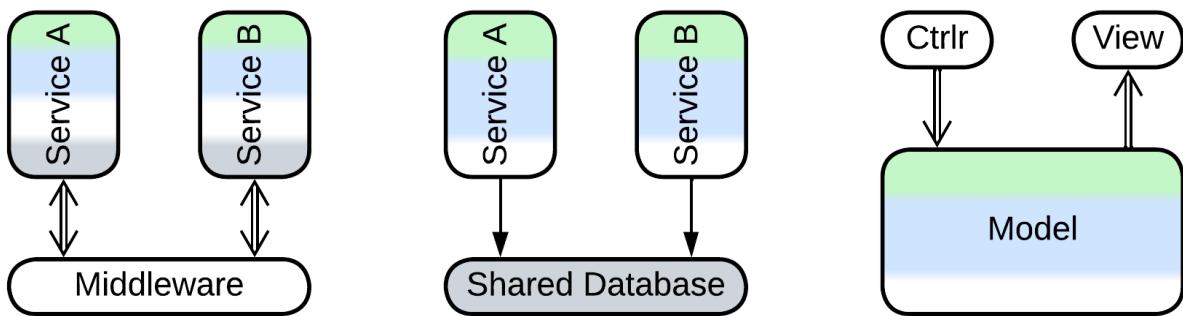
An example of metapatterns

Let's consider the following structure:



It features two (or more in real life) high-level modules that communicate with/via a lower-level module. Which patterns does it match?

- [Middleware](#) – a software that provides means of communication between other components.
- [Shared Database](#) – a space for other components to store and exchange data.
- [Model-View-Controller](#) – a platform-agnostic business logic with customized means of input and output.



My idea of grouping patterns by structure seems to have backfired – we got three distinct patterns that have similar structural diagrams. The first two of them are related – both implement indirect communication, and their distinction is fading as a *Middleware* may feature a persistent storage for messages while a table in a *Shared Database* may be used to orchestrate services. The third one is very different – primarily because the bulk of its code, that is its *business logic*, resides in the lower layer, leaving the upper-level components a minor role.

Notwithstanding, each of the patterns we found is a part of a distinct cluster:

- [Middleware](#) is also known as (*Message Broker* [[POSA1](#), [POSA4](#), [EIP](#), [MP](#)] and is an integral part of *Message Bus* [[EIP](#)], *Service Mesh* [[FSA](#)], *Event Mediator* [[FSA](#)], *Enterprise Service Bus* [[FSA](#)] and *Space-Based Architecture* [[SAP](#), [FSA](#)]).
- [Shared Database](#) is a kind of [Shared Repository](#) [[POSA4](#)] (*Shared Memory*, *Shared File System*), and the foundation for *Blackboard* [[POSA1](#), [POSA4](#)], *Space-Based Architecture* [[SAP](#), [FSA](#)], and *Service-Based Architecture* [[FSA](#)]).
- [Model-View-Controller](#) [[POSA1](#), [POSA4](#)] is a special kind of [Hexagonal Architecture](#) (aka *Ports and Adapters*, *Onion Architecture* and *Clean Architecture*) which itself is derived from [Plugins](#) [[PEAA](#)] (*Addons*, *Plug-In Architecture* [[FSA](#)]), or the misnomer *Microkernel Architecture* [[SAP](#), [FSA](#)]).

Our touching on a single geometry of structural diagrams revealed a web of 20 or so pattern names that spreads all around. With such a pace there is a hope of exploring the whole fabric which is known as *pattern language* [[GoF](#), [POSA1](#), [POSA2](#), [POSA5](#)].

There are three lessons to learn:

- The distribution of business logic is a crucial aspect of structural diagrams.
- Metapatterns are interrelated in multiple ways, forming a pattern language.

- Each metapattern includes several well-established patterns.

What does that mean

Chemistry has the [periodic table](#). Biology has the [tree of life](#). This book strives towards building something of that kind for software and systems architecture. You can say “That makes no sense! Chemistry and biology are empirical sciences while software architecture isn’t!” Is it?

Part 1. Foundations

This part defines some ideas which are used throughout the book. Please feel free to skip (through) it as you probably know most of them quite well.

Modules and complexity

This chapter is loosely based on [A Philosophy of Software Design](#) by John Ousterhout and [my article](#).

Any software system which we encounter is very likely to be too complex to comprehend all at once – the human mind is incapable of discerning a large number of entities and their relations. It tends to simplify reality by building abstractions: as soon as we define the many shiny pieces of metal, glass and rubber as a ‘car’ we can identify ‘highways’, ‘parking spaces’ and ‘passengers’ – we live in a world of the abstractions which we create. In the same way the software we write is built of services, processes, files, classes, procedures – modules that conceal the swarm of bits and pieces which we are powerless against. Let’s reflect on that.

Concepts and complexity

Any system is comprised of *concepts* – notions defined in terms of other concepts. For example, if you are implementing a phonebook, you deal with *first* and *last names*, *numbers*, *sorting*, and *search*, which one must always keep in mind for any phonebook-related development task – just because requirements for the phonebook are described in terms of those concepts and their relations.

In the code high-level concepts are embodied as services, modules or directories while lower-level concepts match to classes, API methods or source files.

Concepts are important because it is their number (or the number of the corresponding classes and methods) that defines the *complexity* of a system – the cognitive load which developers of the system face. If the programmers grasp the behavior of a component they work on in detail they tend to [become extremely productive](#) and are often able to find [simple solutions for seemingly complex tasks](#). Otherwise the development is slow and requires extensive testing because the programmers are [unsure of how their changes affect the system's behavior](#).

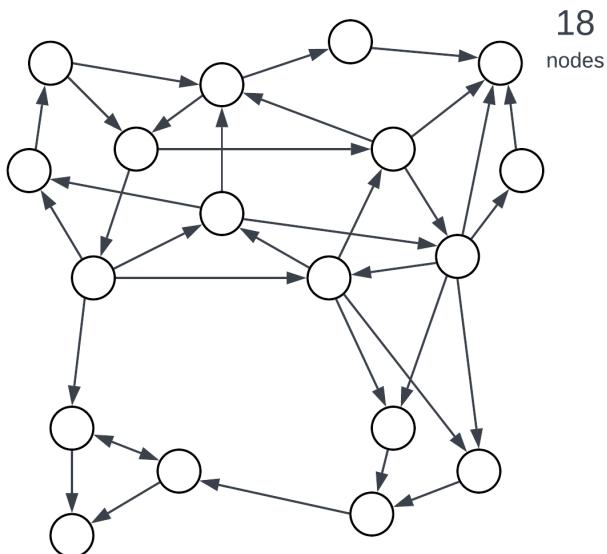


Figure 1: Complexity correlates with the number of entities.

Modules, encapsulation and bounded context

Let's return to our example. As you implement the phonebook you find out that sorting and search are way more complex than you originally thought. Once you prepare to enter the international market you are in [deep trouble](#). Some telephony providers send 7-digit numbers, others use 10 digits, still others – 13 digits (with either "+" or "0" for the first character). German has "ß" which is identical to "ss" while Japanese uses two alphabets simultaneously. Once you start reading standards, implementing all the weird behavior and responding to user complaints you feel that your phonebook implementation is drowning in the unrelated logic of foreign alphabets' special cases. You need *encapsulation*.

Enter *modules*. A module wraps several concepts, effectively hiding them from external users, and exposes a simplified view of its contents. Introducing modules splits a complex system into several, usually less complex, parts.

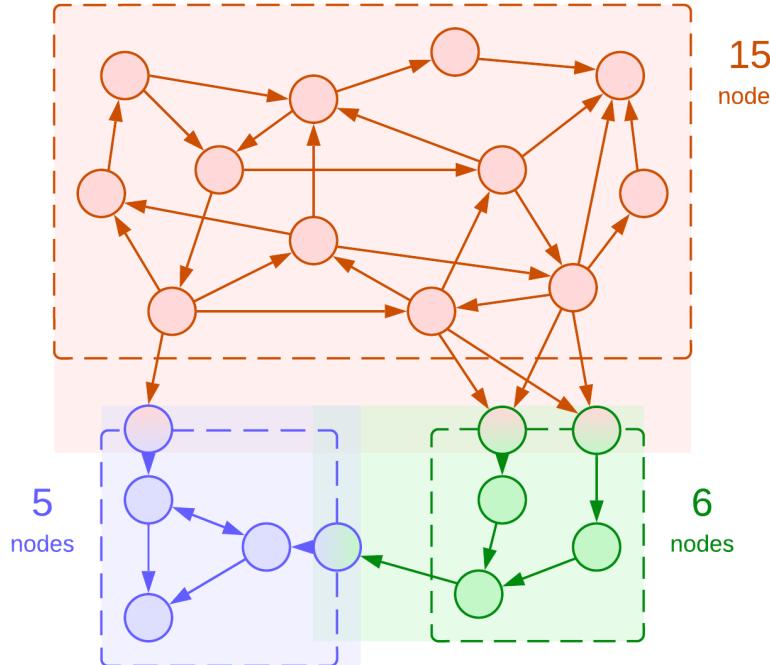


Figure 2: Dividing a system into modules, bounded contexts highlighted.

This diagram has several important points to note:

- Modules create new concepts for their *public APIs*.
- The API entry points add to the complexity of *both* the owner module and its clients.
- The total number of concepts in the system has increased (from 18 to 22) but the highest complexity in the system has dropped (from 18 to 15).

Here we see how introducing modularity applies the [divide and conquer](#) approach to lessen the cognitive load of working on any part of a system at the cost of a small increase in the total amount of work to be done.

In our phonebook example the peculiarities (including case sensitivity) of the locale-aware string comparison and alphabetical sorting of contact names should better be kept behind a simple string comparison interface to relieve the programmer of the phonebook engine of the complexity of supporting foreign languages.

Modules represent *bounded contexts* [DDD] – areas of the knowledge about a system that operate distinct sets of terms. In the case of phonebook the *collation* and *case sensitivity* do not matter for the phonebook engine – they are defined only in the context of language support. On the other hand, *matching a contact by number* is not defined in the language support module – that term exists only in the phonebook engine. It is the complexity of the current bounded context that a programmer struggles with.

Apart from dividing the problem into simpler subproblems, modules open the path to a few extra benefits:

- *Code reuse*. A well-written module that implements something generic may be used in multiple projects.
- *Division of labor*. Once a system is split into modules and each module is assigned one or more programmers, development is efficiently parallelized.
- *High-level concepts*. Some cases allow for merging several concepts of the original problem into higher-level aggregates, further reducing the complexity:

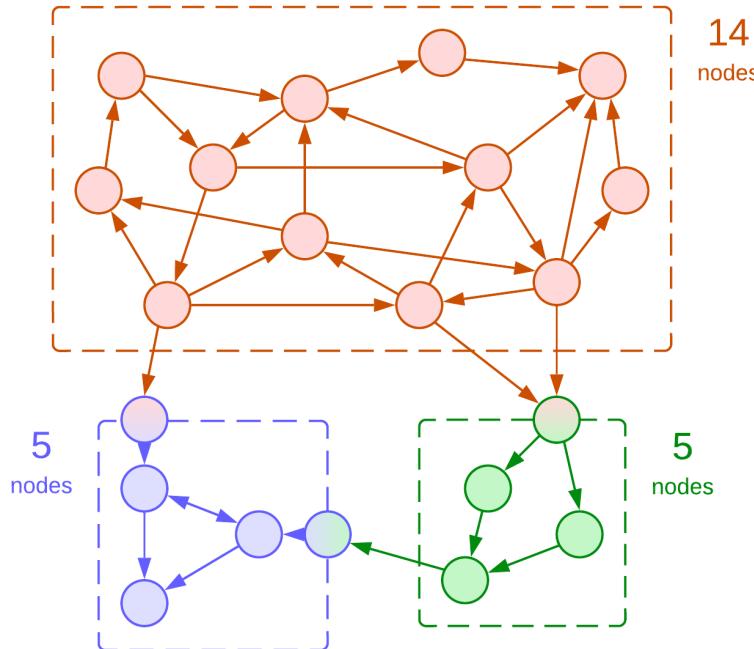


Figure 3: Merged two API concepts in the green module.

For example, the original definition of a phonebook contained *first name* and *last name*. Once we separate the language support into a dedicated module, we may find out that various locales differ in the way they represent contacts: some (USA) use ‘first name + last name’ while others (Japan) need ‘last name + first name’. If we want to abstract ourselves from that detail, we should use a new concept of *full name* which conjoins first and last names in a locale-specific way. Such a change actually simplifies some of the phonebook’s representation logic and code as it replaces two concepts with one.

Coupling and cohesion

We need to learn a couple of new concepts in order to use modules efficiently:

Coupling is a measure of the number (density) of connections between modules relative to the modules’ sizes.

Cohesion is a measure of the number (density) of connections inside a module relative to the module’s size.

The rule of thumb is to aim for *low coupling and high cohesion*, meaning that each module should encapsulate a cluster of related (intensely interacting) concepts. This is how we have split the system in figures 2 and 3. Now let’s see what happens if we violate the rules:

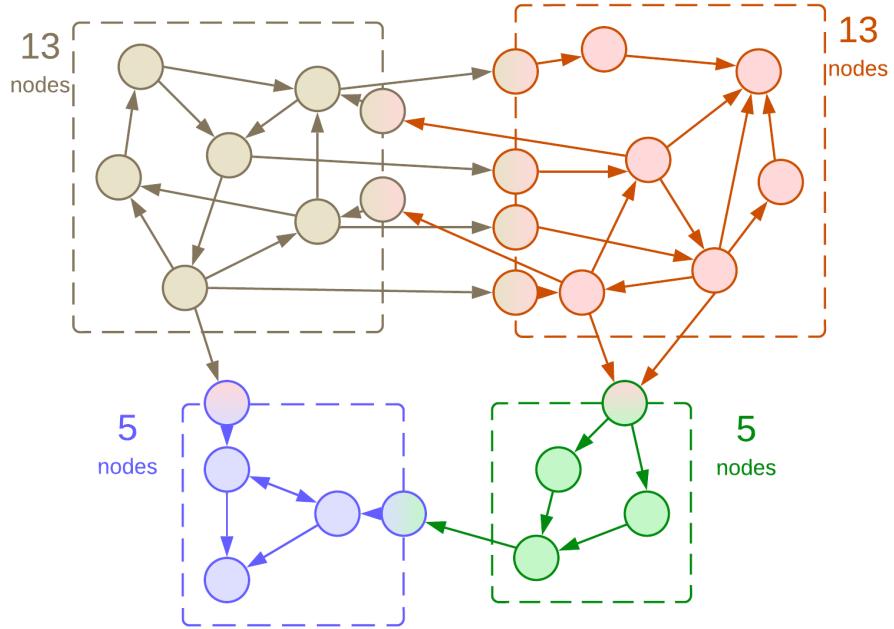


Figure 4: The upper modules are tightly coupled.

Splitting a cohesive module (a cluster of concepts that interact with each other) yields two strongly coupled modules. That's what we wanted, except that each of the new modules is nearly as complex as the original one. Meaning, that we now face two hard tasks instead of one. Also, the system's performance may be poor because communication between modules is rarely optimal, and we've got too much of that.

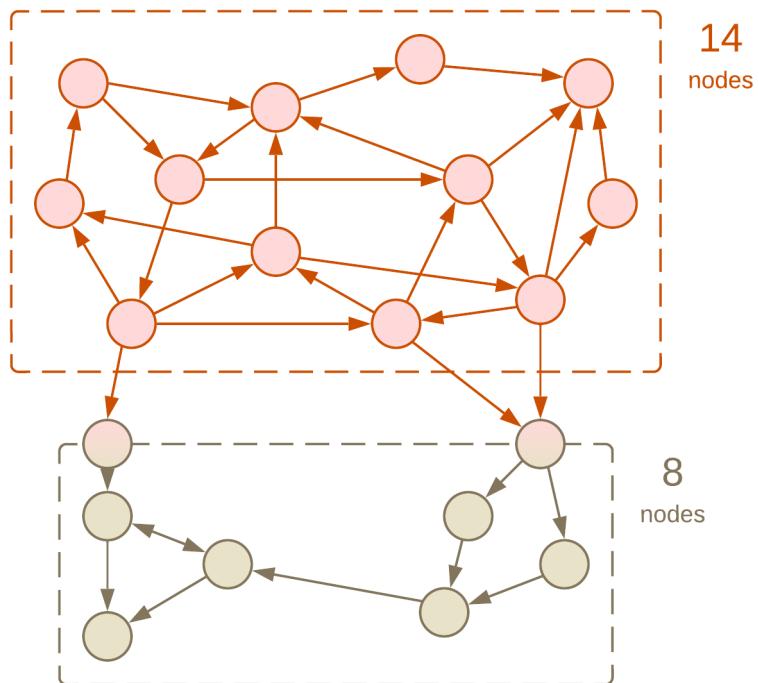


Figure 5: The lower module has low cohesion.

What happens if we put several clusters of concepts in the same module? Nothing too evil happens with small modules – the module gets a higher complexity than each of its

constituents, but lower than their sum. In practice, multiple unrelated functions are often gathered in a ‘utils’ or ‘tools’ file or directory to alleviate *operational complexity*.

Development and operational complexity

What we discussed above is *structural* or *development complexity* – the number of concepts and rules inside a bounded context. However, we also need to understand operations and components of the system as a whole, leading to *operational* or *integration complexity*:

- Does this new requirement fit into an existing module or does it call for writing a dedicated one?
- Which libraries with known security vulnerabilities do we use?
- Is there any way to cut our cloud services cost?
- 1% of requests time out. Would you please investigate that?
- My team needs to implement this and that. Do we have something fit for reuse?
- What the **** is [that global variable](#) about?
- Do we really need this code in production?
- I need to change the behavior of that shared component a little bit. Any objections?

When there are hundreds or thousands of modules deployed nobody knows the answers. That’s similar to the case of one needing to do something in Linux: hundreds of tools are pre-installed and thousands more are available as packages, but the only real way forward is first searching the web for your needs, then trying two or three recipes from the results to see which one fits your setup. Unfortunately, Google does not index your company’s code.

Composition of modules

A module may encapsulate not only individual concepts, but even other modules. That is not surprising as an OOP class is a kind of module – it also has public methods and private members. Hiding a module inside another one removes it from the global scope, decreasing the operational complexity of the system – now it is not the system’s architect’s responsibility but the responsibility of the maintainer of the outer module who cares about the inner module. On one hand, that builds a manageable hierarchy in both the organization and the code. On the other hand, code reuse and many optimizations become nearly impossible as internal modules are hardly known organization-wide:

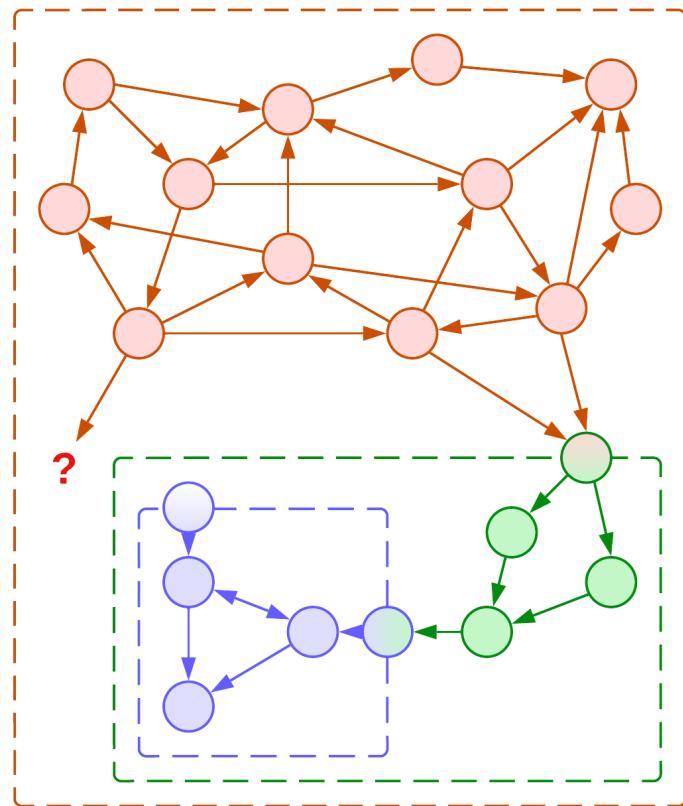


Figure 6: Composition of modules prevents reuse.

If the functionality of our internal module is needed by our clients, we have two bad options to choose from:

Forwarding and duplication

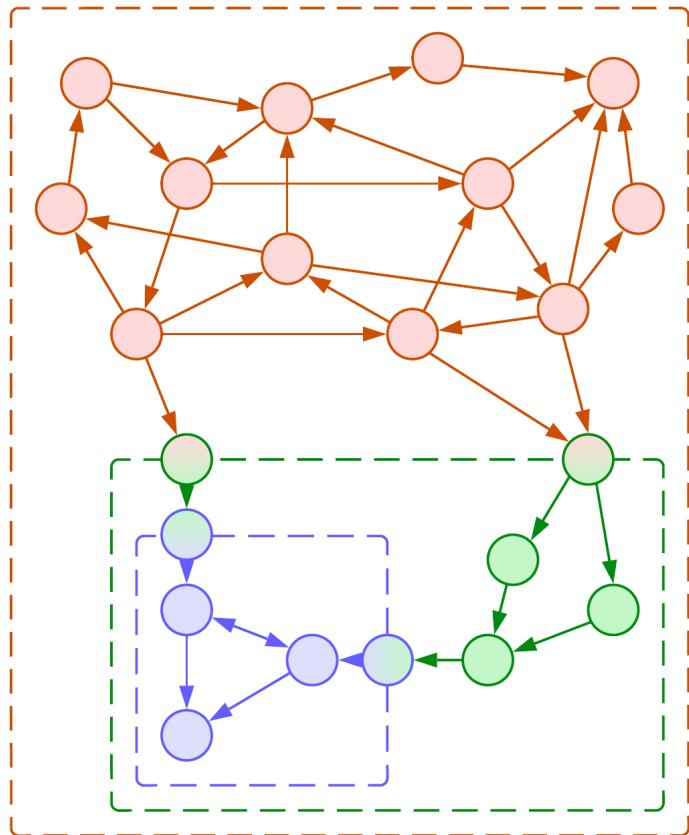


Figure 7: Forwarding the API of an internal module.

We can add the API of a module which we encapsulate to our public API and forward its calls to the internal module. However, that increases the complexity and lowers the cohesion of our module – now each client of our module is also exposed to the details of the methods of the module we have encapsulated even if they are not interested in using it.

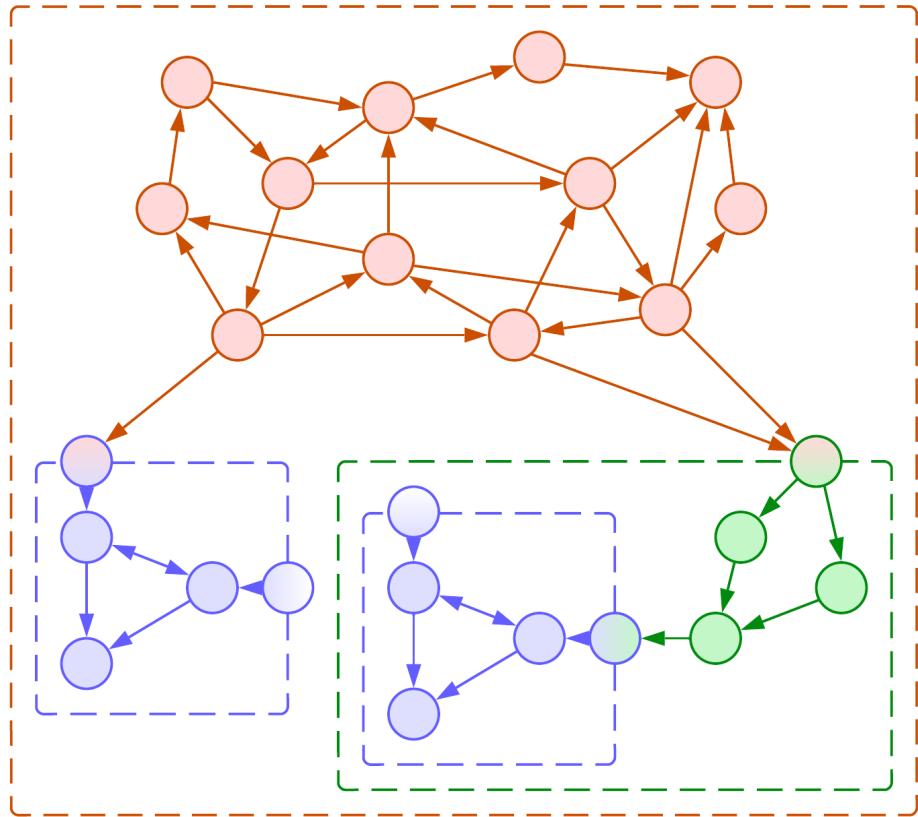


Figure 8: Duplicating an internal module.

Another bad option is to let the clients that need a module which we encapsulate duplicate it and own the copies as their own submodules. This relieves us of any shared responsibility, lets us modify and misuse our internals in any way we like, but violates [a couple of rules](#) of common sense.

Both approaches, namely keeping all the modules in the global scope and encapsulating utility modules through composition, found their place in history [[FSA](#)]. [Service-Oriented Architecture](#) was based on the idea of reuse but fell prey to the complexity of its [Enterprise Service Bus](#) which had to account for all the interactions (API methods) in the system. In response, the [Microservices](#) approach turned the tide in the opposite direction: its proponents disallowed sharing any resources or code between services to enforce their decoupling.

Summary

Complexity is the number of *concepts* and their relations which one must remember to work efficiently. A *module* hides some of the concepts from its users but creates new concepts (its *interface*). *Coupling* is the measure of dependencies between the modules, while *cohesion* is the same for the concepts inside a module. We prefer *low coupling and high cohesion* to group related things together.

Having too many modules causes trouble for the system's maintainers. A module may contain other modules. When a client wants to use a submodule, the wrapping module may extend its interface to forward client's requests to the submodule or the client may deploy a

copy of the submodule for its own use. Both approaches gave rise to prominent architectures.

Forces, asynchronicity, and distribution

Many systems rely on asynchronous communication between their components or are distributed over a network. Why is dividing a system into modules or classes then not enough in real life?

Requirements and forces

Any software is built to meet a set of (explicit or implicit) *requirements*. As a bare minimum, you as a programmer must have at least a vague vision of how your software is expected to operate. At the most, business analysts bring you volumes of incomprehensible documentation they wrote for the sole purpose of forcing you to practice [[DDD](#)].

Some requirements are *functional*, others are *non-functional*.

Functional requirements describe what the system must do: a night vision device must be able to represent heat radiation as a video stream; a multiplayer game must create a shared virtual world for users to interact with over a network; a tool for formatting floppies ... er, formats floppies.

[*Non-functional requirements*](#), also known as *forces*, define expected properties of the system and are known to drive architectural decisions [[POSA1](#), [POSA5](#)]. They may be formulated or implied: our game should be fast enough and stable enough. A medical application should be extremely well-tested. An online shop should provide an easy way to add new goods. Notice all those “fast enough”, “stable enough”, “well”, and “easy” terms on the wishlist. Sometimes they form an [*SLA*](#) with numbers: your service should be available 99.999% of the time.

Let's take an example.

A night vision surveillance camera may spend seconds compressing its video stream to limit the required network bandwidth – this kind of system sacrifices low latency in favor of low traffic. The device will need a fast CPU (probably a DSP) and lots of RAM to store multiple frames for efficient compression.

A night vision camera of a drone should have moderately low latency as the drone (and probably its operator) uses the video stream for navigation. Thus it should send out every frame immediately, except that it may still spend some time compressing the frame to JPEG to achieve a balance between latency and bandwidth. Pushing for extremely low latency of the camera does not help much because the whole system is limited by the delay of the radio communication and the human operator in the loop.

Night vision goggles or helmets are [*stringent on latency*](#) to the extent which no ordinary digital system satisfies, thus [*expensive analog devices*](#) have to be used.

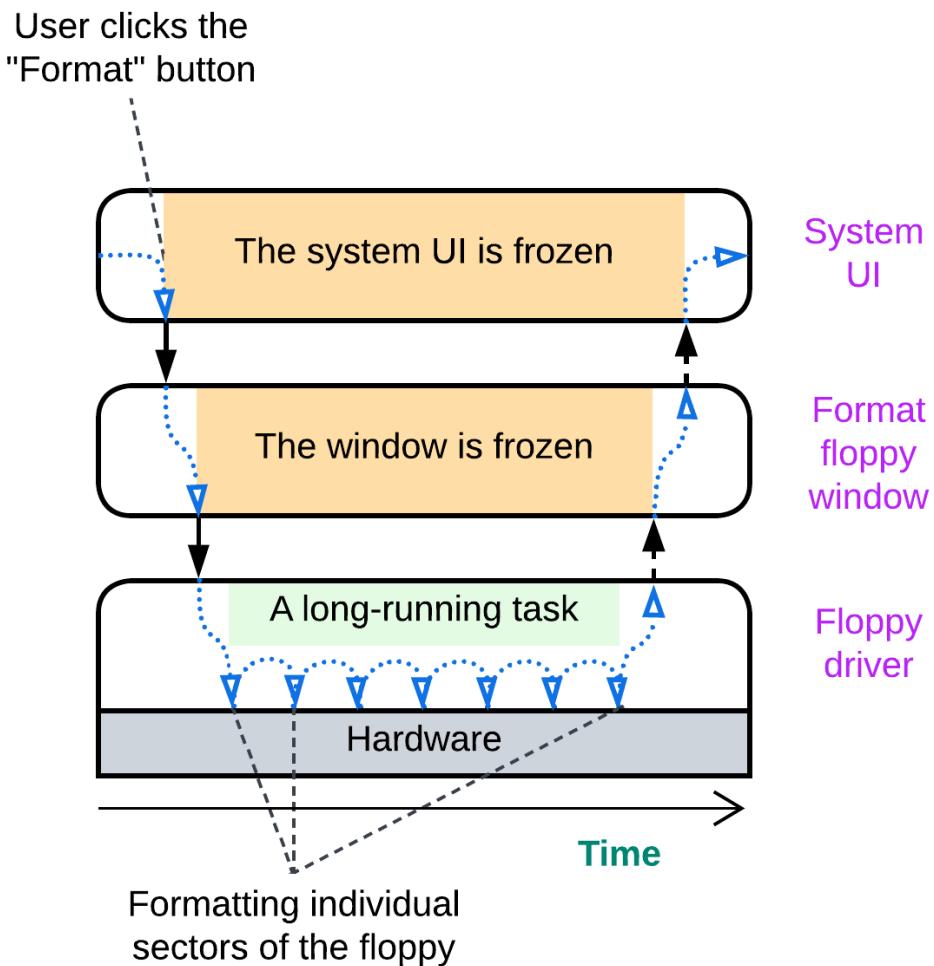
Here we see how non-functional requirements – namely, latency, bandwidth and cost – impact all the stuff all the way down to hardware. The same happens with multiplayer games: while a chess client is a simple web page, a fighting tournament or a first-person shooter is very likely to need a client-installed application that processes much of the game logic locally while relying on a highly customized network protocol to decrease communication latency.

Another example is the choice of programming language: you can quickly write your system in Java or Python sacrificing its performance or you can spend much more time with C or C++ and manual optimization to achieve top performance at the cost of development speed.

Conflicting forces

We see that forces influence architecture. That becomes way more interesting when a system is shaped by conflicting forces – the ones that, though opposing each other, still need to be met by the architecture.

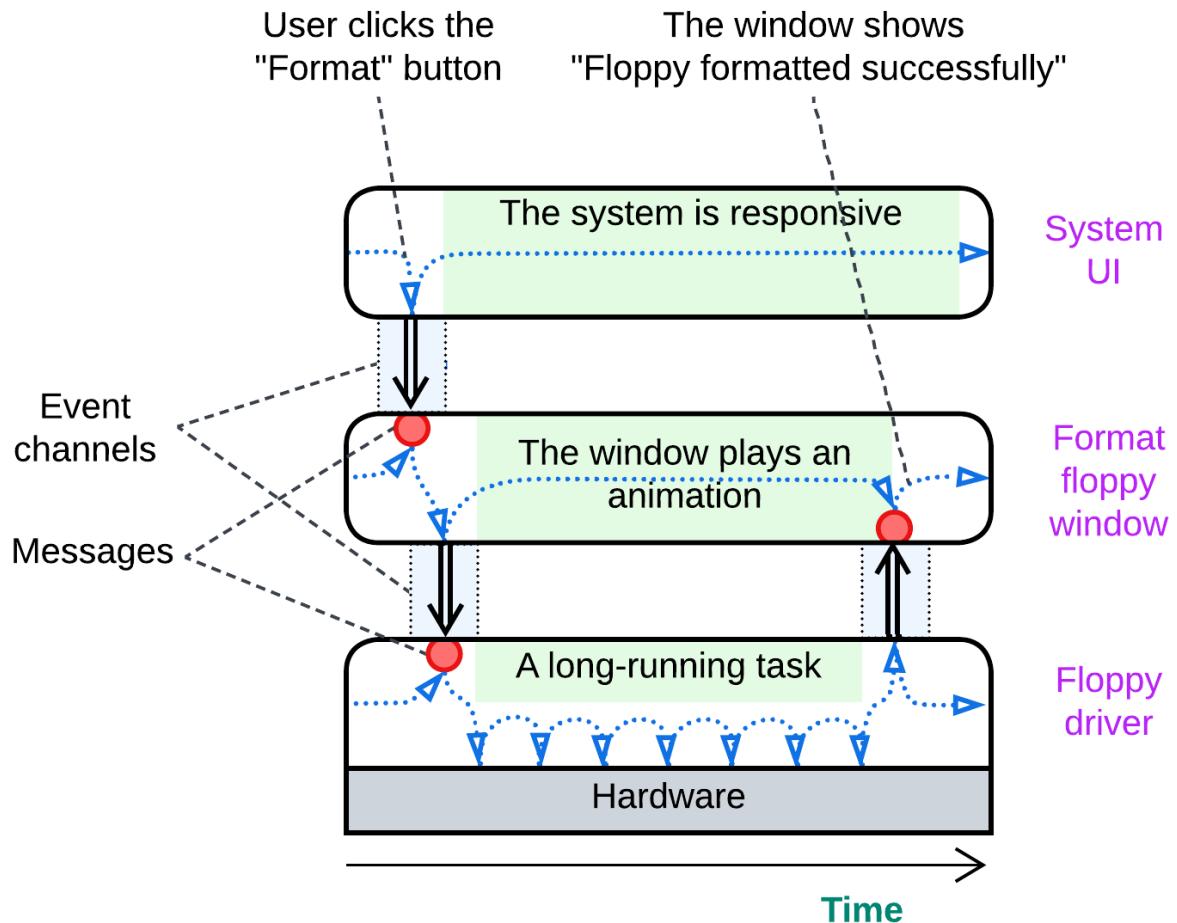
Remember how old Windows used to freeze on formatting a floppy or when it encountered one with a bad cluster? Let's see how such things could have happened (though [the real cause was a bit different](#), it also came from the modules' sharing a context).



The system implements the function it was made for – it formats floppies. However, while the low-level module is busy interacting with the hardware, all the modules above it have no chance to run because they have called the driver and are waiting for it to return. The modules are there, with the code separated into bounded contexts (the UI does not need to care about sectors and FATs) but all of them share non-functional properties – latency in this case. Either the UI is responsive or the floppy driver runs a long-running action. We need the UI and the driver to execute independently.

Asynchronous communication

If the modules cannot communicate directly (call each other and wait for the results returned) how should they interact? Through an intermediary where one of the modules leaves a message for another. Such an intermediary may be a message queue, a pub/sub channel, or even a data record in shared memory. The sender posts its message and continues its routine tasks. The receiver checks for incoming messages whenever it has a free time slot. Behold multithreading in action!



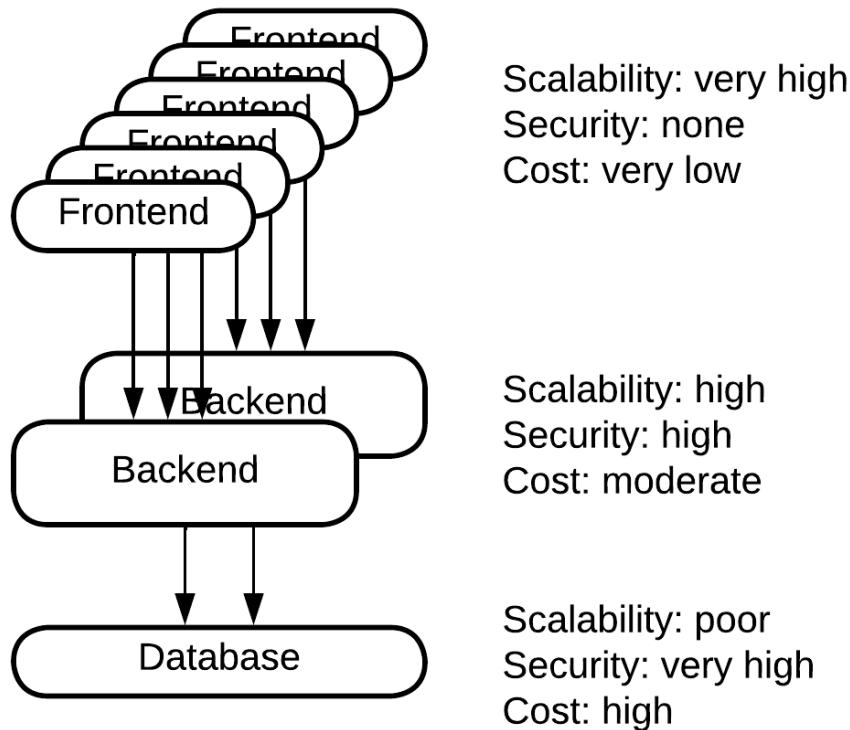
Distribution

Once modules run independently, we can separate them into processes and even distribute the processes over multiple computers. That is required to address fault tolerance and high availability and solve conflicts around scaling or locality.

Consider a web site. Most of them follow [Three-Tier Architecture](#):

- A *frontend* runs in users' browsers.
- A *backend* runs on the business owner's servers.
- A *database* usually runs on a single powerful server.

This common division makes quite a lot of sense:



Websites are accessed by many users simultaneously. Any business owner wants to pay less for his servers, thus as much work as possible is offloaded to the users' web browsers which provide unlimited resources for free (from the business owner's viewpoint). Here we have a nearly perfect scalability – the business owner pays only for the traffic.

Other parts of the software are business-critical and should be protected from hacking. Such ones are kept on private servers or in a cloud. This means that the business owner pays for the servers while they may scale their application by flooding it with money.

The deepest layer – the database – is nontrivial to scale. Distributed databases are expensive, consume a lot of traffic, and still scale only to a limited extent. It often makes more sense to buy or rent top-tier hardware for a single database server than to switch over to a distributed database.

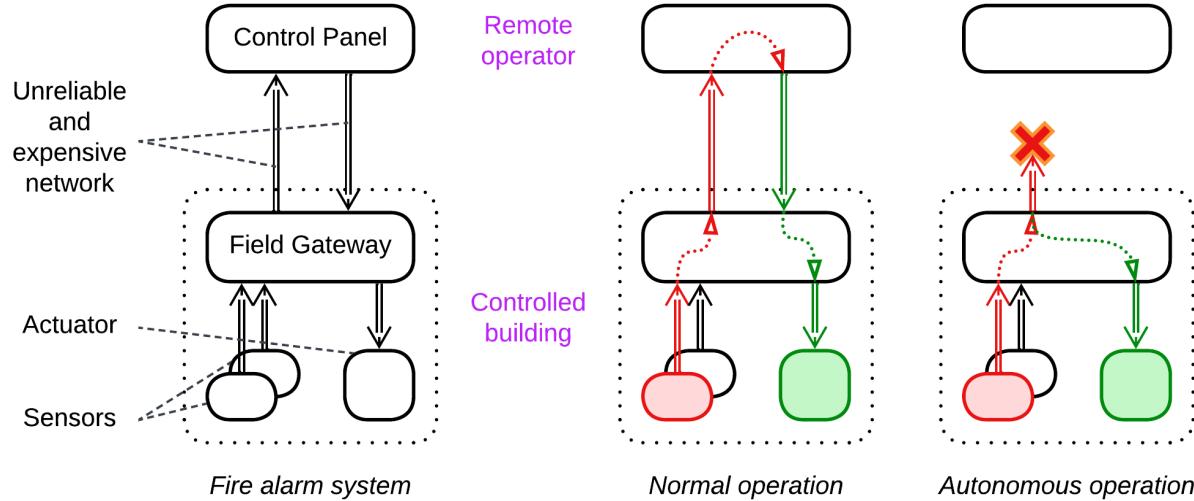
This is a good example of how the physical distribution of a system solves the scalability, security, and cost conflict by choosing the best possible combination of forces for each module. Whatever is not secure scales for free. Whatever does not scale gets assigned expensive hardware. Whatever remains is in between.

Another example comes from IoT – a fire alarm system. They tend to use 3 tiers as well:

- Sensors (smoke or fire detectors) and actuators (fire suppression, sirens, etc.).
- A *field gateway* – a kind of router the sensors and actuators are directly connected to.
- A *control panel* – some place where operators drink their coffee.

Sensors and actuators are cheap and energy-efficient but dumb devices. They do not react to events unless explicitly commanded. The control panel is where all the magic happens, but it may be unreachable if the network is damaged or the wireless communication is jammed. Field gateways stand in between: they collect information from the sensors, aggregate it to save on traffic, communicate with the control panel, and even activate actuators if the control panel is unreachable. In this case a part of the business logic is installed in the dedicated devices which are located within the controlled building.

Here reliability conflicts with accuracy: a human operator makes an accurate estimate of the threat and chooses an appropriate action, but it is not granted that we can always reach the operator. Thus to be reliable we add an inaccurate but trustworthy fallback reaction.



A similar pattern can be found with robotics, drones or even computer hardware (e.g. a HDD): dedicated peripheral controllers supervise their managed devices in real time while a more powerful but less interactive central processor drives the system as a whole.

The goods and the price

Let's review what we found out.

Modules make it easier to reason about the system, enable development by multiple teams in parallel, and resolve some conflicts between forces. For example, development speed against performance or release frequency against stability are solved by choosing a programming language and release management style on a per module basis.

The cost is the loss of some options for performance optimization between modules and the extra cognitive load while debugging a module you are not familiar with.

Asynchronous communication is a step forward from modules that solves more conflicts between forces. It addresses latency and multitasking.

We pay for that with context switches and the need to copy and serialize data transferred in messages, making communication between participating modules slower. Debugging asynchronous communication becomes non-trivial as one cannot single-step in the debugger from the message sender into the message handler.

Distribution builds on asynchronous communication (as networks are asynchronous) and decouples participant components in such forces as scalability, security and locality. It separates release cycles of the services involved and makes it possible for the system to recover from failures of some of its components.

The price? Even slower and more complicated communication in the now distributed system (networks are quite laggy and unreliable) and extremely inconvenient debugging as you need to connect to multiple components over the network.

We see that the more isolated our modules become, the more forces are decoupled and the more flexible the resulting system grows. But this very same decoupling devastates the system's performance and makes debugging into a nightmare.

Any moral? There is one, even a few.

- [Do not overisolate](#). Go asynchronous or distributed only if you are *forced* to. Especially if you are actively evolving your system. Especially in an unfamiliar domain.
- Cohesive logic belongs together. If you split it among asynchronous or distributed components, it may become very hard to debug.
- Components that communicate a lot should reside together. Distributing them may kill performance and even break the consistency of the data.

Four kinds of software

Software products vary in their goals which, surprisingly, determine their structure and operation. The main features of software make a fuzzy continuous space rather than a strict set of well-defined classification options. Let's examine two of its dimensions:

Source of inputs

Some systems receive their inputs from users via text commands or UI controls, often mediated by a network protocol. Such inputs are highly meaningful, structured and compact – processing a single user command often invokes a larger part of the program's functionality.

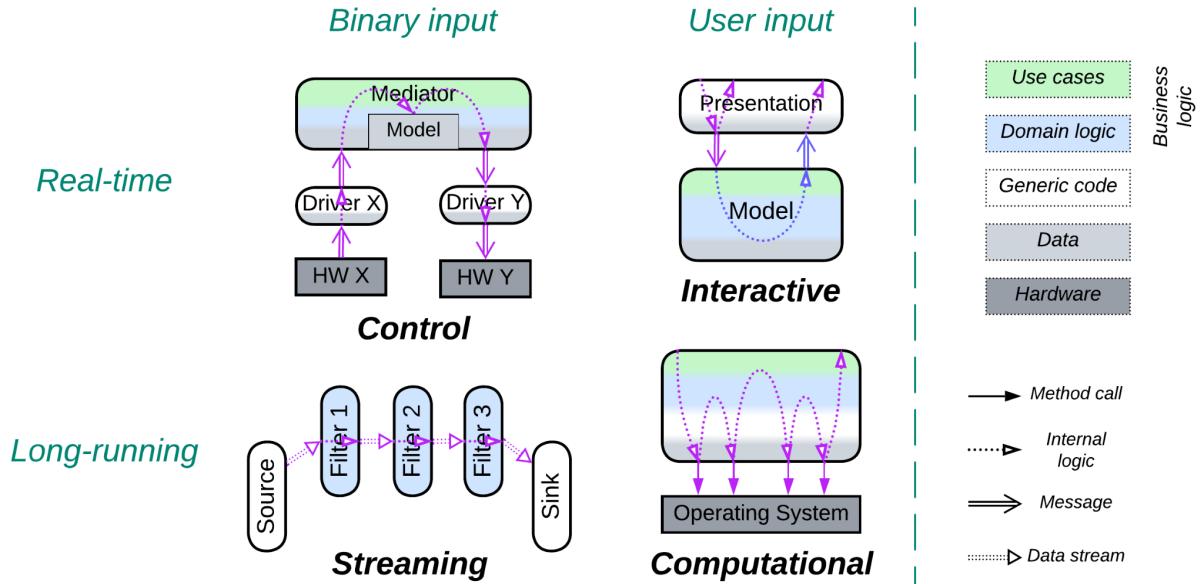
Another kind of system deals with binary data or signals that come from hardware. Those input sources have low informational payload – a digitized movie is orders of magnitude larger than a book written in a human language and still omits many details of the original text. Raw inputs often require context (the program's state) to understand their meaning: the same sequence of bits may be treated as a part of a video, audio, executable file or archive – and the correct interpretation is known only from the command your program is running and the name and header of the file it is processing.

Latency constraints

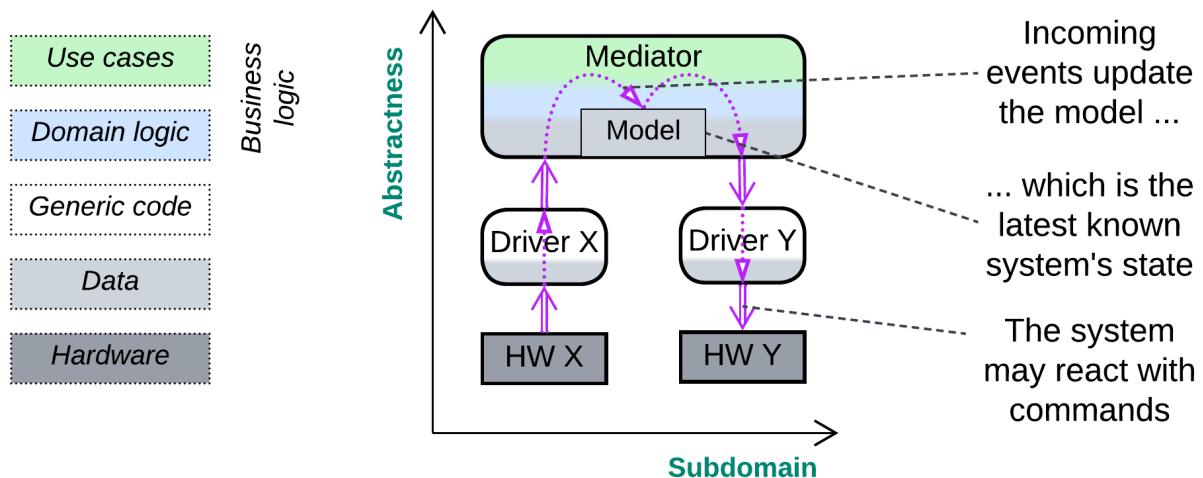
Programs that control hardware or interface with users need to respond to their inputs in real time. Milliseconds of delay result in any kind of bad stuff that ranges from negative reviews and lost business opportunities to lost human lives (which are the same from the business PoV). That kind of software can never block its execution or run long calculations and has to keep all the data involved in memory.

Other programs are not that time-constrained – they run a single task for a long time, accessing many files or consulting remote services in the process. They may use more powerful decision-making algorithms and all the data in the world – but they are slower to respond.

Those dimensions make four corner cases that vary in architectural styles:



Control (real-time, hardware input)



A *control* application supervises several hardware or software interfaces with the goal of keeping a certain system-wide *invariant*:

- A fly-by-wire program receives readings from the airplane's sensors and adjusts its actuators to keep the vehicle on its course, which is the invariant.
- When a telephony gateway receives an outgoing call request from one of the devices that it manages, it checks the dialed number, initiates an incoming call to a matching interface, and connects a voice channel between the devices as soon as the destination accepts the call. The invariant is that calls live in pairs: if a call comes in, it is to be rejected or a derived call should go out of the device.
- A Container Orchestrator checks the health of the services it manages and restarts any one which is slow to respond to its keep-alive request, making sure that all the services are online.

As a control software must react quickly, it has no time to read from storage or obtain from other components the data it needs to make decisions. Thus it has to build and maintain in its memory a *model* of the system it controls:

- An autopilot knows the last measured plane's coordinates, speed, angles and states of all its actuators.
- A telephony application models both the phones and accounts it supervises and the calls present in the system.
- A container manager remembers the state of every service it oversees, the time of the last health check and user request statistics (number of requests and processing time).

When a program receives information from a component it controls, it updates its in-memory model with the new data and checks if the target invariant still holds:

- A plane should remain on its course, otherwise its [angles](#) or thrust must be adjusted.
- If there is a new incoming call, a VoIP gateway must create a corresponding outgoing call.
- If a service is known to have not responded for a while, it must be killed and restarted.

After the compensating action is initiated, the model is updated accordingly, and the software resumes processing events. After a while it should receive events that confirm that the invariant was restored:

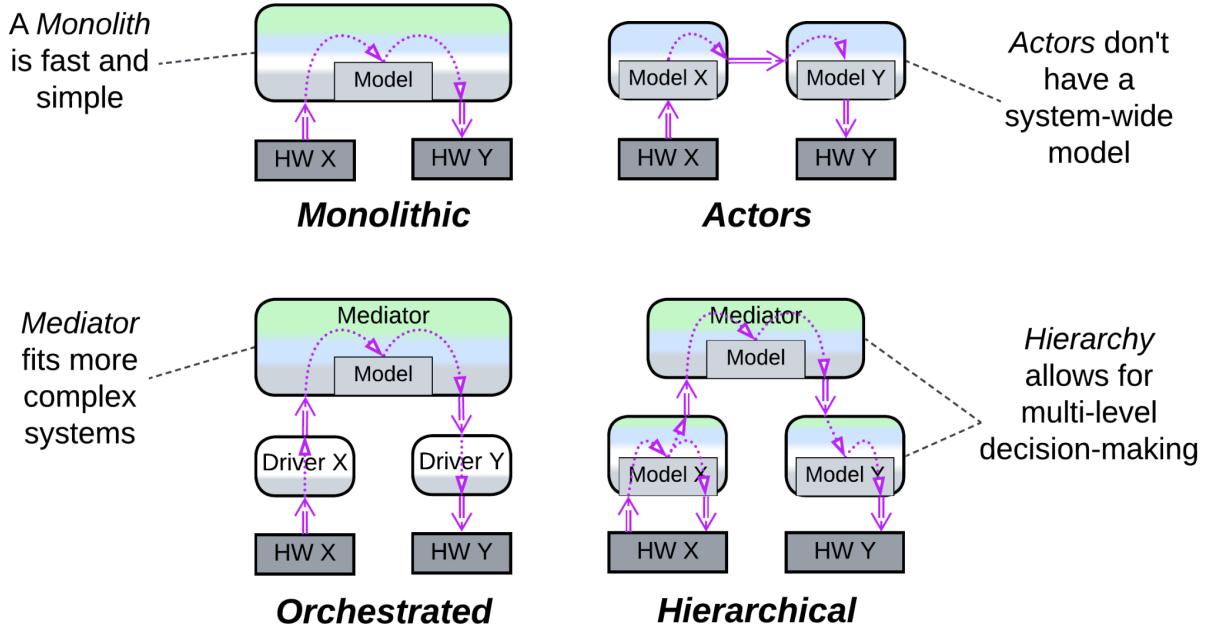
- The new sensor readings will show that the plane is back on its course.
- The outgoing call will have been accepted and the gateway will connect the voice path between the call parties.
- The container will have been restarted successfully and will be serving user requests.

The flow of changes through the system is **M-shaped**: it starts as a low-level event, gets interpreted by a hardware driver or protocol stack, and goes up to an [Orchestrator](#), which updates its model and decides if there should be any reaction, which would then go all the way down to the same or another low-level interface.

Variants

At the architectural level, control systems are [event-driven](#) – their components communicate by messages. There are several architectures I am aware of:

- [Monolith](#) is the simplest and fastest implementation, usually tiny enough to run everything in a single [super loop](#).
- Plain ([choreographed](#)) [Actors](#) fit systems of trivial logic but massive scale, like messenger backends. Each actor models the component (remote device or user) it interacts with, and there is no global model except for a registry of actors.
- [Orchestrated modules](#) are a better solution for complicated ([cohesive](#)) systems that need centralized management. The [Orchestrator](#) contains the whole system's model and integration logic.
- A [Hierarchical Orchestrator](#) can manage even more complex systems. The [Orchestrator](#) may run synchronously with polymorphic specialized components or asynchronously. In the last case each sub-orchestrator reacts independently based on its own model but also sends a notification to the high-level component which builds a global strategy and programs the smaller models of sub-orchestrators.



Patterns

The following patterns are prominent in control software:

- [**Actors**](#) – partitioning the domain into self-consistent asynchronous entities allows fine control over the order of execution of system activities, provided that we run a [preemptive scheduler](#) (from [POSIX real-time threads](#) or [RTOS](#)) – we can assign top priority to reading data from communication interfaces (which would quickly overflow if the data is not retrieved), make reacting to events a bit less urgent, and still have leftovers of our CPU time for such long-running tasks as file access or strategic planning.
- [**Proactor** \[POSA2, POSA4\]](#) – almost every component is single-threaded, reactive, and non-blocking, which makes the system very responsive. The downside is being unable to represent a multi-step scenario as a single function, which is usually unimportant as no predefined scenario ever survives event-driven reality unshattered. Following a planned path leads directly to your grave. Proceed stepwise, checking for dangers every millisecond, being ready to jump away from any approaching trouble.
- [**Mediator** \[GoF\]](#) (a kind of [**Orchestrator**](#)) – when you rely on information from and manage several devices or interfaces, you need a single entity that knows what is going on, makes informed decisions and dispatches commands to be executed. It integrates all the lower-level components into a coherent system. It is a *Mediator*.
- [**Hexagonal Architecture**](#) – hardware components quickly become obsolete, and if you want your software to survive for a decade, you must be able to change them at will. And if you want to run, test, and debug your code on your desktop, you often need to [stub](#) or [mock the hardware](#).
- [**Hierarchy**](#) – if you manage a variety of interfaces that have the same role (telephony protocols, account providers, or payment systems) you want to make them polymorphic towards your main application logic. In other cases, like [IoT](#), you may need to start reacting immediately with little precision and correct your actions after you have spent more time on better planning, which is achieved through a hierarchy of feedback loops.

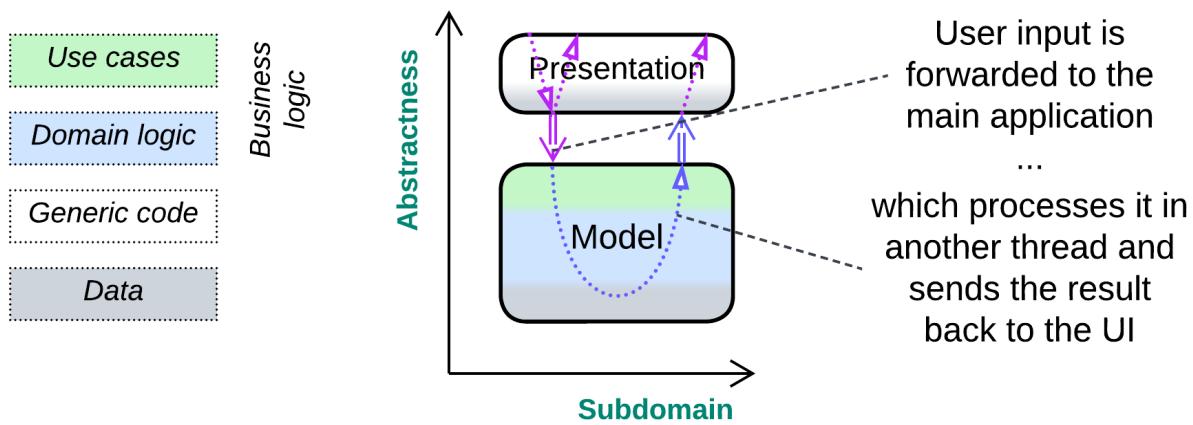
Implementation

In the code we may (or may not) see *State* [GoF] (aka *Objects for States* [POSA4]) close to hardware or protocol interfaces, but the higher-level logic is likely to depend on multiple parameters which would make too many combinations to write down as state classes, thus it tends to be coded as a decision tree instead.

System components (*Actors*) have private in-memory data and communicate only by messages. They are usually single-threaded and non-blocking (*Proactor*) – this way, the only locks in the system are those protecting the *actors*' message queues and the global memory manager, which don't contain anything complicated to block on for a noticeable time. And as nothing ever blocks, the whole system is extremely responsive.

Messages may be dispatched through multilevel index arrays or *Visitors* [GoF]. Message queues may be shared (a queue per thread priority) or private (a queue per component). With shared queues the destination of a message must either be resolvable from its type (when there is a single instance of every kind of system component) or stored in the message's header.

Interactive (soft real-time, user input)



An *interactive* software deals with users who expect it to provide immediate feedback to their actions. Examples include:

- Desktop and mobile applications, from text editors to browsers.
- Simple games, like chess or tetris.
- UI of embedded devices, such as clocks or air conditioning systems.

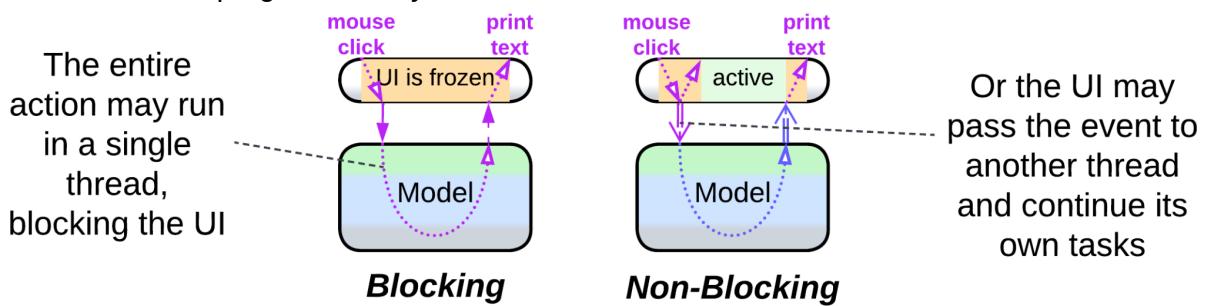
The user interface layer (*Separated Presentation*) receives input (mouse, keyboard, or touchscreen events), interprets it as a user action based on the current application's *state* (pressing enter in a text editor may break a line of text in two, begin a new line or even activate a menu item), forwards the action to the lower levels of the software and displays any feedback received, producing a **U-shaped** flow.

Variants

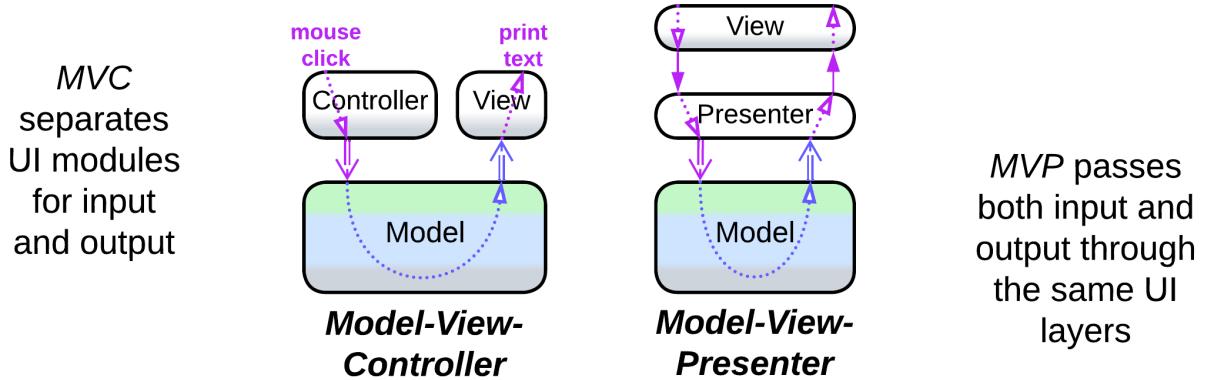
Interactive systems vary in a couple of ways:

- The presentation layer may wait for the business logic to execute the user's action, blocking further user input and screen updates (air conditioner controller), or it may asynchronously pass the command to the lower layer and continue processing new

user input and showing progress of the already running tasks (many games) while the main program is busy with the command.



- There may be dedicated modules for processing user input and output ([MVC family](#) of patterns) or both may pass through the same stack of components ([MVP family](#)). The asymmetric approach deals with raw controller input, which is what most games need, while the bidirectional flow operates UI widgets provided by the host OS or GUI framework.



Patterns

You will likely encounter:

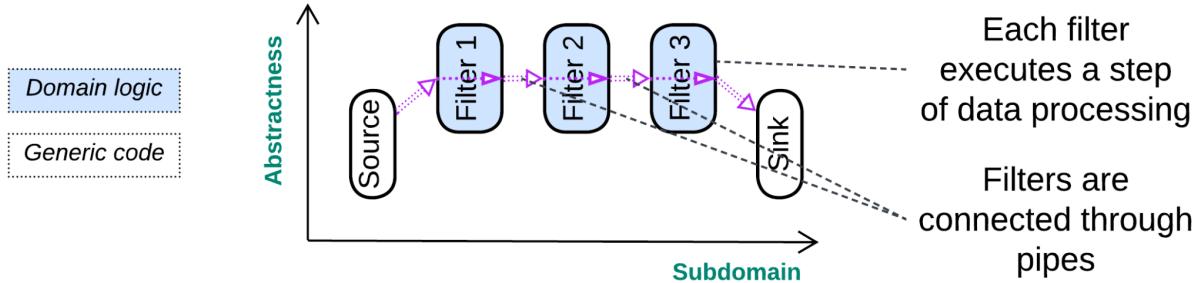
- [Separated Presentation](#) – the business logic is unaware of the implementation of the UI layer, though this may not be the case with some games that rely on *game development frameworks*. This pattern is usually implemented by:
 - [Model-View-Presenter](#) (MVP) – two input layers, namely: *view* which receives input and shows output and *presenter* which translates between the business logic, called *model*, and the view.
 - [Model-View-ViewModel](#) (MVVM) also has two layers, but the intermediate *ViewModel* binds to the view and bears its state.
 - [Model-View-Controller](#) (MVC) [[POSA1](#), [POSA4](#)] separates the input (*controller*) from the output (*view*).
- [State](#) [[GoF](#)], subclassed by [[POSA4](#)] into *Objects for States*, *Methods for States*, and *Collections for States*, is prominent in games, though it may not always be implemented explicitly.
- [Flyweight](#), [Command](#), [Observer](#) and many other [[GoF](#)] patterns originated with desktop software and [may often appear](#) in games.

Implementation

The presentation layer may be called into by the desktop environment or it may run in a dedicated thread, for example, to play animations.

The business logic is likely to rely on its own threads, at least for long-running actions. The presentation would usually subscribe to updates from the business logic.

Streaming (continuous, raw data input)



A *streaming* system processes a long sequence of similar events or data packets, usually by transforming individual items in a predetermined way:

- Each party in an audio call or video conference deals with incoming and outgoing media streams. For example, incoming audio stream processing involves: saving audio packets to a [jitter buffer](#) to restore their order, [compensation for lost packets](#), decoding, equalization, and playback. Outgoing audio passes through the following steps: [echo suppression or cancelation](#), noise reduction, equalization, encoding, adding network headers, and sending packets to the network.
- An image recognition system applies a [long sequence of transformations](#) to every input image.
- Hardware often works with streams. For example, a [CPU instruction pipeline](#) comprises at least: instruction fetching, instruction decoding and register fetching, execution, memory access, and register writeback. High-end processors may have up to 20 stages.
- Many UNIX command-line tools process streams of lines of text, allowing for complex text processing by chaining pre-existing utilities [[DDIA](#)].

Streaming usually passes data through a chain of transformations ([pipeline](#)) which differ in functionality but stay at about the same level of abstractness – there are no managers or hierarchy. Such a --shaped structure allows for all the specialized components to deal with their chunks of the stream in parallel, which increases the system's throughput but suffers from moderate to high latency.

Variants

As stream-processing [Pipelines](#) can exploit multiple CPU cores and specialized hardware, they are found everywhere from lowest-level firmware to large-scale distributed systems. [[DDIA](#)] classifies them into:

- [Stream processing](#), where the pipeline is always alive waiting for more input to come.
- [Batch processing](#), where the pipeline runs till it finishes processing an input file.

Patterns

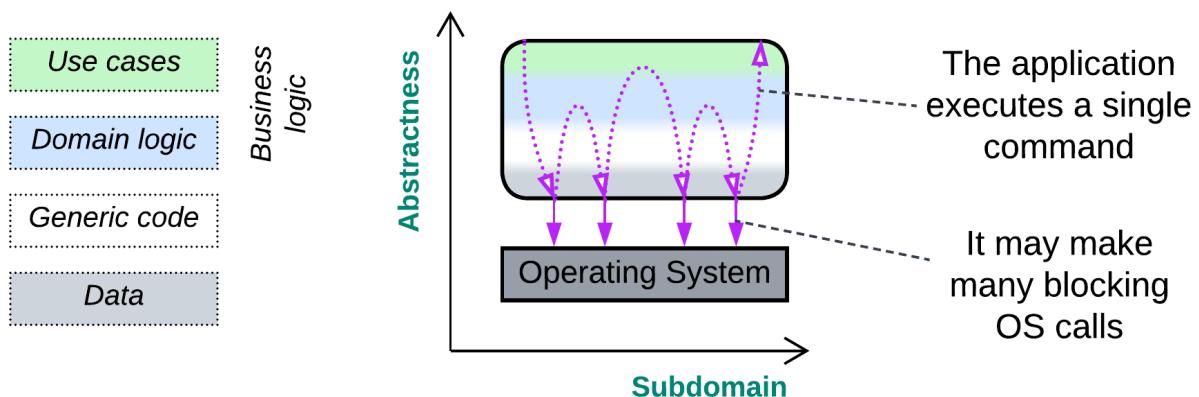
- [Pipes and Filters](#) [[POSA1](#), [POSA4](#)] is the stream processing pattern that describes pipelines: the system is built of *filters* (individual data processing steps) connected through *pipes* (data channels).

- [Choreographed Event-Driven Architecture](#) (EDA) [[SAP](#), [FSA](#), [DDS](#)] and [Data Mesh](#) [[LDDD](#), [SAHP](#)] are tree-like pipelines that process streams of domain events or data, correspondingly.
- [\[EIP\]](#) is full of patterns for the distributed processing of event streams.

Implementation

Every filter is likely to run in its own thread and be unaware of other filters – it only knows where to pull its input from and push its output into. One common challenge is to [slow down](#) a too fast data producer or scale data consumers when too much intermediate data accumulates in the pipe between them.

Computational (single run, user input)



Finally, there is a large group of applications created to process long-running commands:

- A scientific calculation runs for days or weeks to provide insight into the physical reality or validate a new theory.
- A compiler creates a platform-specific binary by processing text files with the program's code.
- An interpreter runs scripted actions on its user's behalf.

In each case the application starts with parsing (interpreting) its input, proceeds to execute it in a stepwise (and likely looped) manner and finishes by outputting results of the run, making a W-shaped flow.

Variants

Some computational systems are single-use with a hard-coded task (calculation of Pi) while others can execute a variety of user scenarios (script interpreters).

Patterns

Long-running programs with user input are probably the most common, ancient, and well-studied kind of software, which also inspired many design patterns. Those of special significance are:

- [Interpreter](#) [[GoF](#)] that supports very complex user commands.
- [Facade](#) [[GoF](#)] or [Process Manager](#) [[EIP](#)] (kinds of [Orchestrator](#)) that executes a user command as a sequence of calls to lower-level components.

Implementation

You likely have some kind of *parser* (complex for SQL or very simple for command-line parameters) that transforms user input into a [syntax tree](#) or a set of flags, correspondingly. Then there is a kind of *main loop* which iteratively executes actions, pre-defined or encoded in the parsed tree, until an *exit condition* is met or the entire input has been processed.

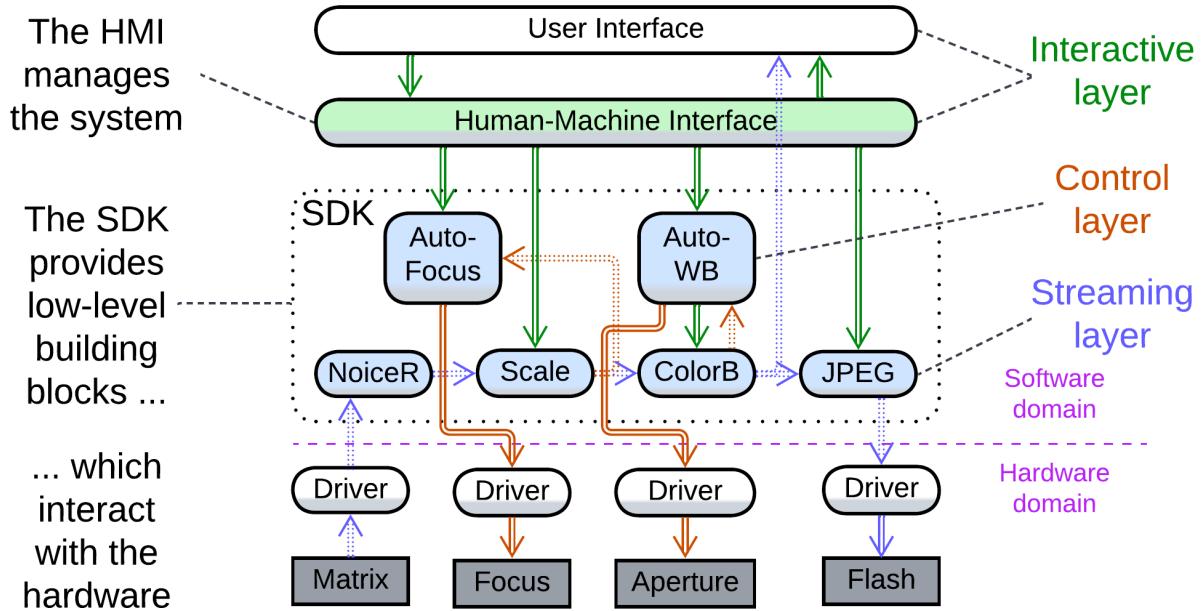
Any calls to external components (OS or libraries) are likely to be blocking as the computation does not need to react quickly to any additional input – actually, it does not read any input or change its behavior for the entire duration of its run.

Parts of computations may be offloaded to [SIMD](#) or [GPU/TPU](#) because that greatly speeds up number crunching which is often characteristic of long-running calculations.

Mixed cases

Most real-life software is too complex to fit the classification outlined above. It tends to merge the paradigms either by mixing them to find a middle ground or by implementing two or three of them at once. Let's inspect a few random examples to see how that happens:

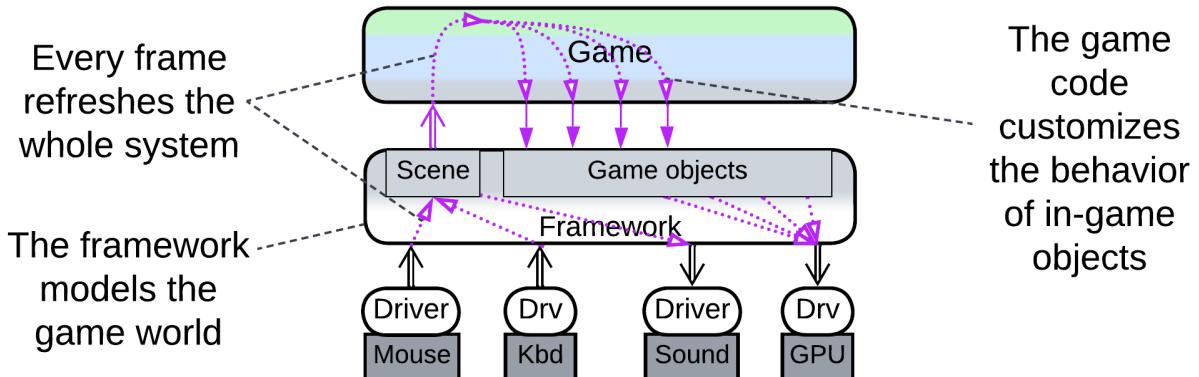
Camera



A digital camera incorporates subsystems of different kinds:

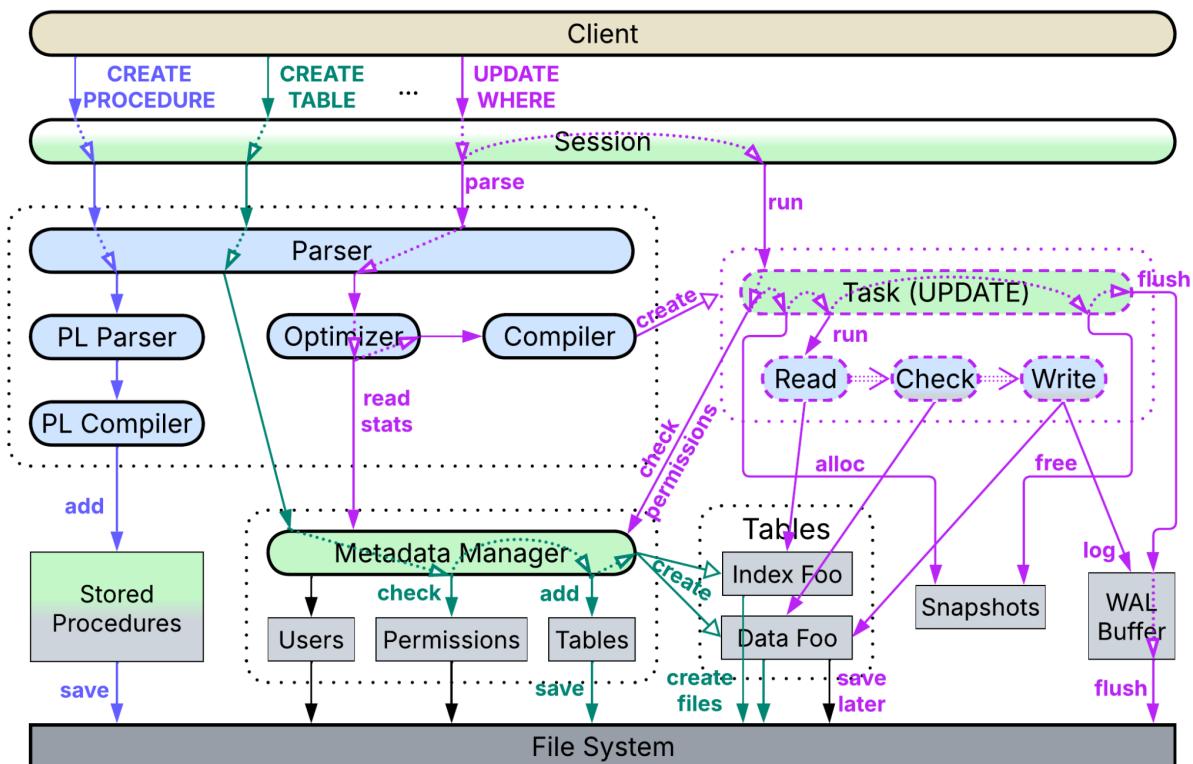
- There is an interactive user interface that receives commands for other components and displays either the video stream from the matrix or the camera's settings menu.
- A control layer provides feedback loops to keep the camera focused on a selected object and preserve the overall brightness level and color balance when shooting in automatic mode.
- An image processing pipeline applies noise reduction, rescaling, and color correction, then either passes the resulting frames to the UI to show them on the screen or proceeds with compressing the frame and storing it as a file.

3D action game



Games with 3D graphics often bypass the host OS' [desktop environment](#) and access the underlying hardware drivers to achieve fine control and improved performance. Such applications, though pretending to be interactive software driven by user input, strongly resemble control systems by polling hardware with fixed frequency (the game's frame rate).

SQL database



SQL databases support several kinds of user commands:

- [Data definition](#) and [data control](#) languages (DDL / DCL) manage the database's metadata: table definitions and user permissions, correspondingly.
- [Data manipulation](#) and [data query](#) languages (DML / DQL) write to and read from the tables.
- [Procedural language](#) (PL) programs user-defined functions (*stored procedures*).

Each kind of command is processed in a unique way:

- When the parser recognizes a DDL or DCL request, it calls a corresponding method that reads or modifies the metadata. If a table is to be altered, the action will either lock the table for the duration of operation or require a complicated workaround to allow other sessions to access the table while it is being restructured.
- DML or DQL input is compiled into a tree of elementary operations (reads, conditions, joins, etc.) which is then passed to the *query optimizer* to be rearranged into a linear sequence of operations that accounts for table sizes and index types. The execution of the resulting *pipeline* depends on its type:
 - As a *query* (DQL) does not change anything, it merely [reserves a virtual snapshot](#) of the current state of the tables, runs the compiled pipeline on the *snapshot* (probably taking quite a while), streams the output to the client and, finally, releases the snapshot.
 - *Commands* (DML) modify data, thus they involve a few extra steps. A snapshot is allocated as well, however, now it is writable, storing all the changes to the database the command's pipeline makes. Every change is also written to a [Write-Ahead Log \(WAL\) buffer](#). When the pipeline or a wrapping transaction completes, the database engine ensures that the data changed in the snapshot is still unchanged in the main database. In case of a conflict the snapshot and WAL buffer are dropped, a new snapshot is allocated on top of the conflicting changes in the main storage, and the command is re-applied. As soon as there are no conflicts, the WAL buffer is flushed to the *WAL file*, which is the single source of truth for the database's crash recovery. After that the changes from the snapshot are integrated into the main data storage, marking any updated data structures as *dirty*. The snapshot and WAL buffer are released, the command returns the number of rows it changed to the user, and another background thread lazily flushes the dirty data to the file system or network storage.

Snapshotting ([MVCC](#)) allows long-running queries ([OLAP](#)) to return consistent data (without reporting any concurrent changes from commands) and commands ([OLTP](#)) to commit with [no need to lock](#) database records.

- PL requests are identified by keywords and forwarded to a PL-dedicated parser which checks their syntax, compiles the user-defined functions into some kind of pseudocode and stores them for further use by queries or commands.

Here we have:

- Dynamically created *streaming pipelines* that represent DQL and DML requests.
- Heavy reliance on parsers and compilers, like in *computational* software.
- A long-running main application that deals with multiple user requests that often need to be executed quickly, like in *interactive* applications.

[[DDIA](#)] is dedicated to the implementation of databases, which are indeed way more complex and varied than what I outlined above.

Summary

We can discern four kinds of systems that differ in their goals, architecture, and code:

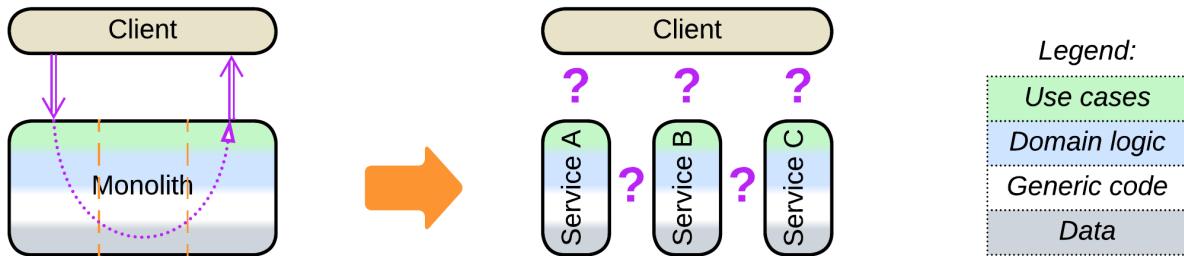
- *Control* – supervises hardware and must react with extremely low latency. It never blocks and relies on an in-memory model of the system it manages.
- *Interactive* – deals with users, should handle their actions and provide feedback in near real time. Its user-facing layer tends to be separated from the main logic.

- *Streaming* – processes boring sequences of data items. Such systems are usually assembled from single-purpose components.
- *Computational* – performs a long-running calculation. It parses user input to understand its task, then goes deep into its engine to execute it and, finally, pops up the result.

Complex real-world software usually involves two or three of these approaches.

Arranging communication

As a project grows, it tends to become subdivided into services, modules, or whatever you call the components that match its subdomains (or *bounded contexts*, if you prefer the [DDD] convention). Still, there remain system-wide use cases that require collaboration from many or all of the system's parts – otherwise the components don't make a single system. Let's see how they can be integrated.



As integration is not unique to distributed systems – it is present even in smaller programs that need to make data, functions, and classes work together – we'll take a look at programming and architectural paradigms next.

Programming and architectural paradigms

Sharing a database is the greatest sin when you architect [Microservices](#) yet [Space-Based Architecture](#) is built around shared data. How do these approaches coexist? Do *Microservices* make any sense if blatantly violating their rules still results in successful projects?

Another programming paradox holds a clue. There was C. Then there came C++ to kill C. Then we've got Rust to kill C++. Now we have C, C++, and Rust, all of them alive and kickin'.

Technologies are specialized

When a new technology emerges, it must show its superiority over existing mature methods. In most cases that is achieved by specialization. Is a car superior to a donkey? It depends. Probably yes, when there are good roads, plenty of gas and spare parts. A car is narrowly specialized, thus some areas have successfully adopted cars, while others still rely on donkeys.

The same holds true for programming languages and architectures. C is good when you work close to hardware and need complete control over whatever happens in the system. C++ is great at partitioning business logic, but it lost the simplicity of its predecessor. Rust will likely shine in communication libraries, which are often targeted by hackers, though we have yet to see its wide adoption. Hence the usefulness (and choice) of a tool or programming language depends on the circumstances.

Let's turn our attention to your average code. It often mixes together:

- *Object-oriented* programming that divides the application into a tree of loosely interacting pieces.
- *Functional* programming, with the output of one function becoming the input to another, [method chaining](#) included.

- *Procedural* programming, where multiple functions access the same set of data, which also happens inside classes whose many methods operate their private data members.

Each [programming paradigm](#) fits its own kind of tasks. Moreover, the same three approaches reemerge at the system level:

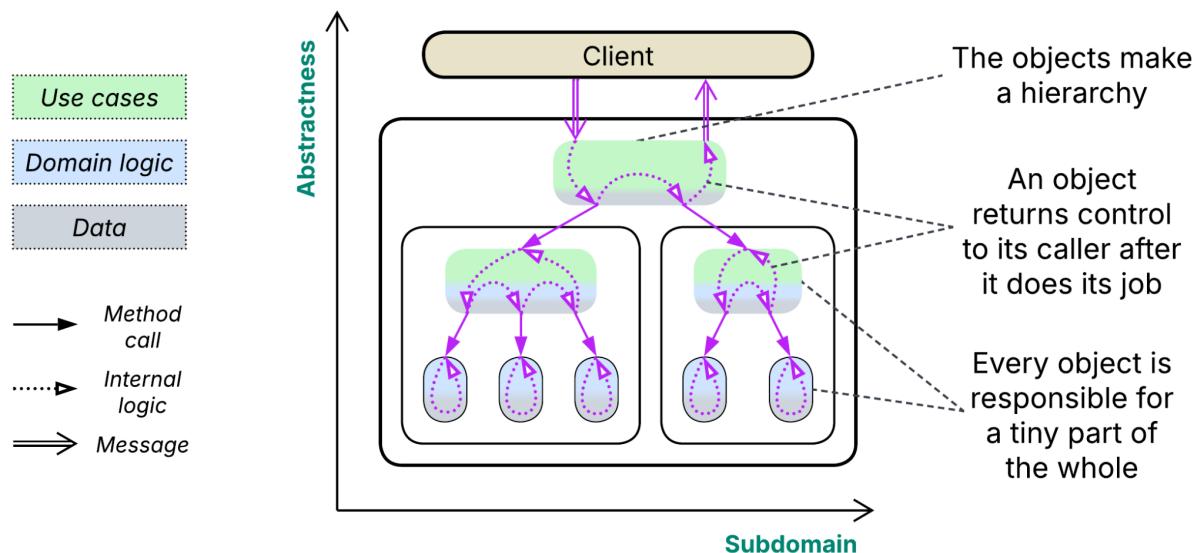
Object-oriented (centralized, shared nothing) paradigm – orchestration

Almost every software project is too complex for a programmer to keep all the details of its requirements and implementation in their mind. Notwithstanding, those details must be written down and run as code.

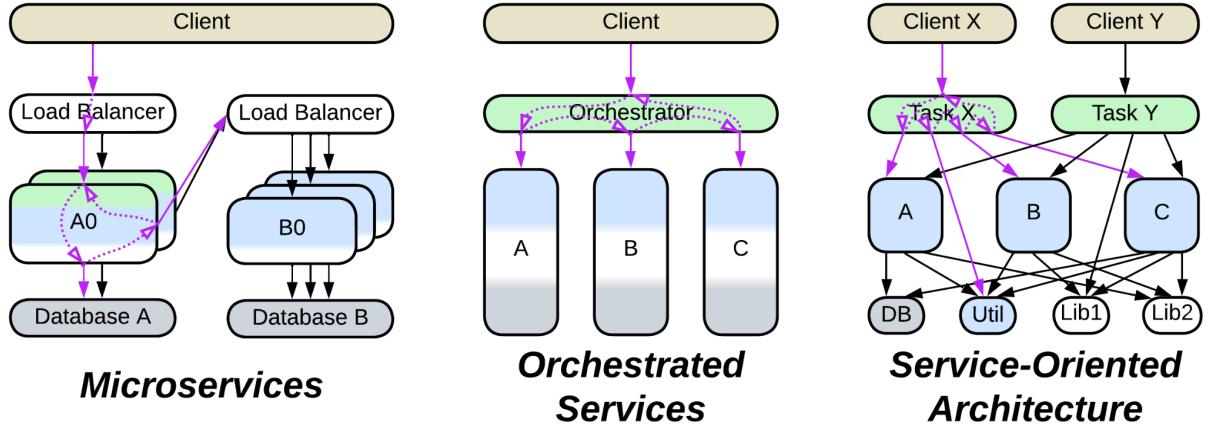
The good old way out of the trouble is called [divide and conquer](#). The global task is divided into several subtasks, and each subtask is subdivided again and again – till the resulting pieces are either simple enough to solve directly or [too messy](#) to allow for further subdivision. Basically, we need to split our domain's *control*, *logic*, and *data* into a single hierarchy of moderately sized components.

We have heard a lot about keeping *logic and data* together: an object (or actor, or module, or service – no matter what you call it) must own its data to assure its consistency and hide the complexity of the component's internals from its users. If the encapsulation of an object's data is violated, the object's code can neither trust nor restructure it. On the other hand, if the data is bound to the logic that deals with it, the entire thing becomes a useful black box one does not need to look into to operate.

Adding *control* to the blend is more subtle, but no less crucial than the encapsulation discussed above. If an object commands another thing to do something, it must receive the result of the delegated action to know how to proceed with its own task. Returning control after the action is conducted enables separation of high-level supervising (orchestration, integration) logic from low-level algorithms which it drives, adding depth to the structure.

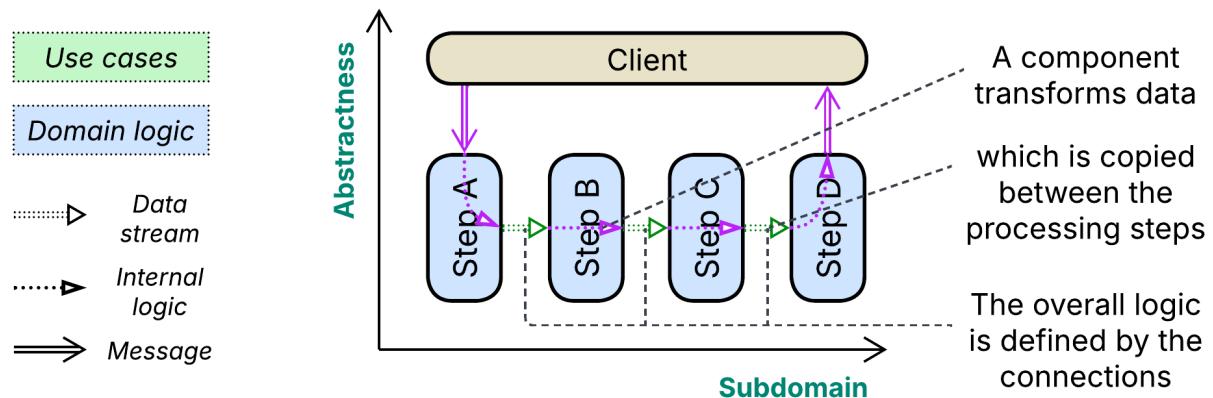


The ability to address complex domains by reducing the whole to self-contained pieces makes object-oriented design ubiquitous. This paradigm, when applied to distributed systems, gives birth to [Microservices](#), [Orchestrated Services](#), and [Service-Oriented Architecture](#).



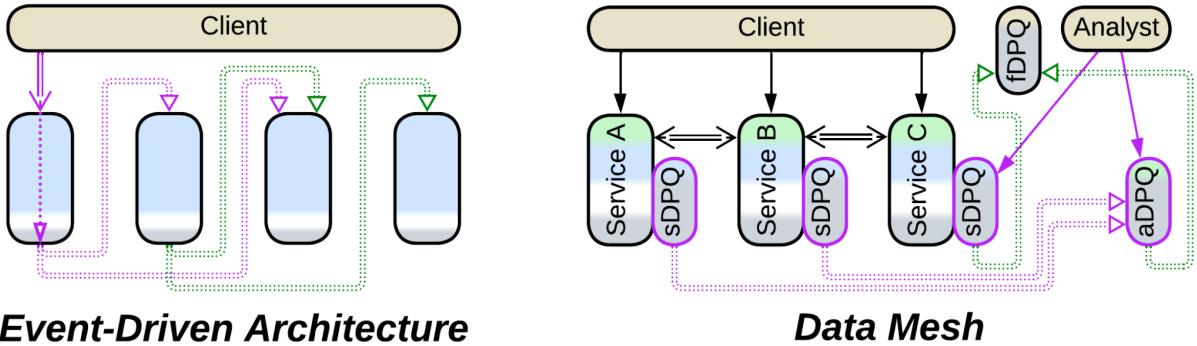
Functional (decentralized, streaming) paradigm – choreography

Sometimes you don't need that level of fine-tuning for the behavior of the system you build – it operates as an [assembly line](#) with high throughput and little variance: its logic is made of steps that resemble work stations along a [conveyor belt](#) through which identically structured pieces of data flow, just like goods on the belt. In that case there is very little to control: if an item is good, it goes further, otherwise it just falls off the line. Here the *control* resides in the graph of connections, the domain *logic* is subdivided, while the *data* is copied between the components.



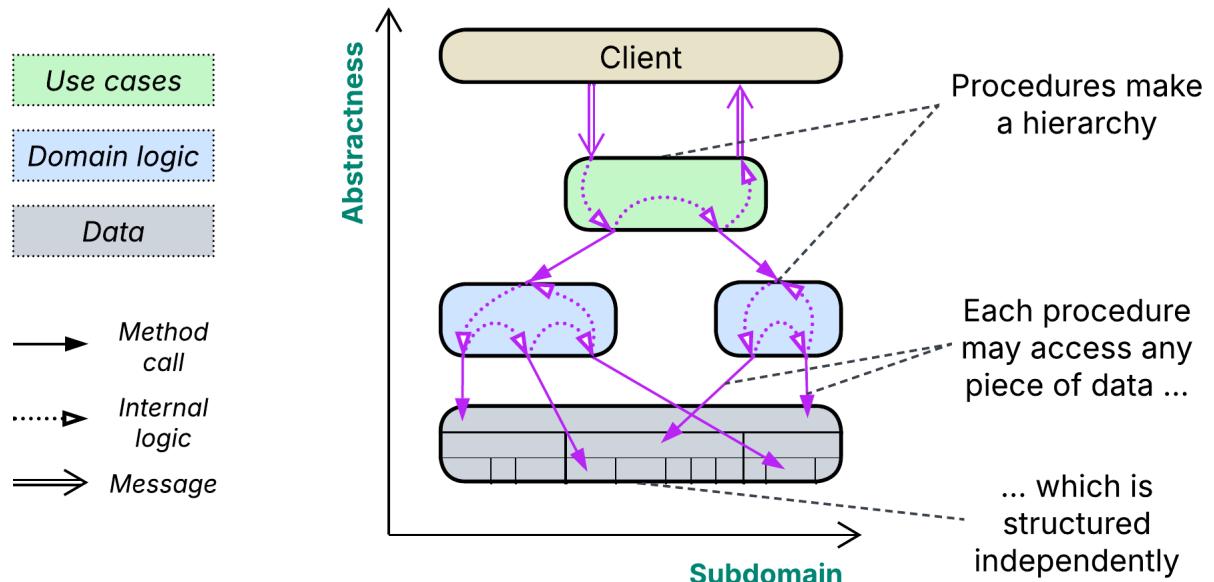
Functional or pipelined design is famous for its simplicity and high performance as the majority of processing steps can be scaled. However, its straightforward application lacks the depth needed for handling complex processes, which would translate into webs of relations between hundreds of functions present at the same level of design. It is also inefficient for choose-your-own-adventure-style ([control](#)) systems where too many too short conveyor belts would be required, negating the paradigm's benefits. And it may not be the right tool for making small changes in large sets of data as you'll likely need to copy the whole dataset between the constituent functions.

In distributed systems the functional paradigm is disguised as [Choreographed Event-Driven Architecture](#), [Data Mesh](#), and various [batch or stream](#) processing [[DDIA](#)] [Pipelines](#).

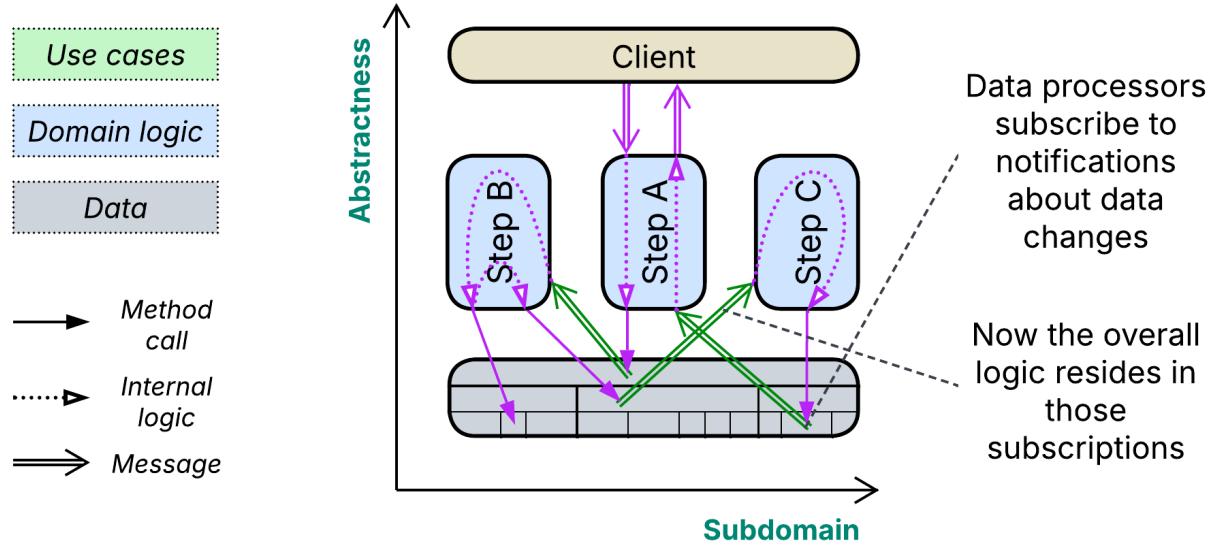


Procedural (data-centric) paradigm – shared data

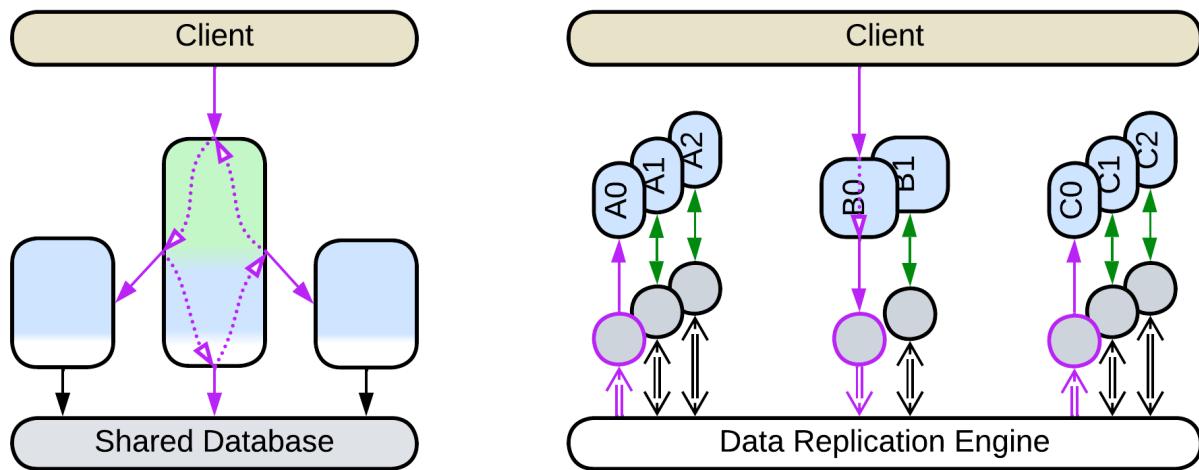
The final approach is integration through data. There are cases where the domain data and business logic differ in structure – you cannot divide your project into objects because each of the many pieces of its logic needs to access several (seemingly unrelated) parts of its data.



In the data-centric paradigm *logic* and *data* are structured independently. In procedural programming, like in object-oriented paradigm, *control* is implemented inside the logic, making the logic layer hierarchical (*orchestrated*). Another, much less common, option relies on *Observer* [GoF] to provide data change notifications, resulting in decentralized (*choreographed*) application logic:



The data-centric approach works well for moderately-sized projects with a stable data model (like reservation of seats in trains or game of chess). The best-known distributed data-centric architectures include [Services with a Shared Database](#) and [Space-Based Architecture](#).



Composite cases

The three programming paradigms tend to collaborate:

- An ordinary class is object-oriented on the outside but procedural inside: each of its methods can access any of its private data members. Moreover, code inside methods may chain function calls, locally applying the functional paradigm.
- [Cell-Based Architecture](#) tends to use *choreography* (pub/sub) between Cells [[DEDS](#)] and *orchestration* or communication via a *shared database* inside them.
- A system of [Services](#) (or [Space-Based Architecture](#)) may be integrated through both [Orchestrator](#) and [Shared Database](#) (or *processing grid* and *data grid*, correspondingly).

Reality is more complex

We have reviewed a few cases directly supported by common programming languages. However, there is a wide variety of possible combinations of (at least) the following dimensions, each making a unique programming paradigm:

- Synchronous (method calls) vs asynchronous (messaging), with closely related:
 - Imperative vs reactive.
 - Blocking vs non-blocking.
- Centralized (orchestrated) vs decentralized (choreographed) flow.
- Shared data (tuple space) vs [shared nothing](#) (messaging).
- Commands (actors) vs notifications (agents).
- One-to-one (channels) vs many-to-one (mailboxes) vs one-to-many (multicast) vs many-to-many (gossip) communication.

Some of the combinations look impossible or impractical, others are narrowly specialized thus uncommon, while many more are commonplace. Discussing all of them would require insights from people who have used them in practice and that would take a dedicated book.

Summary

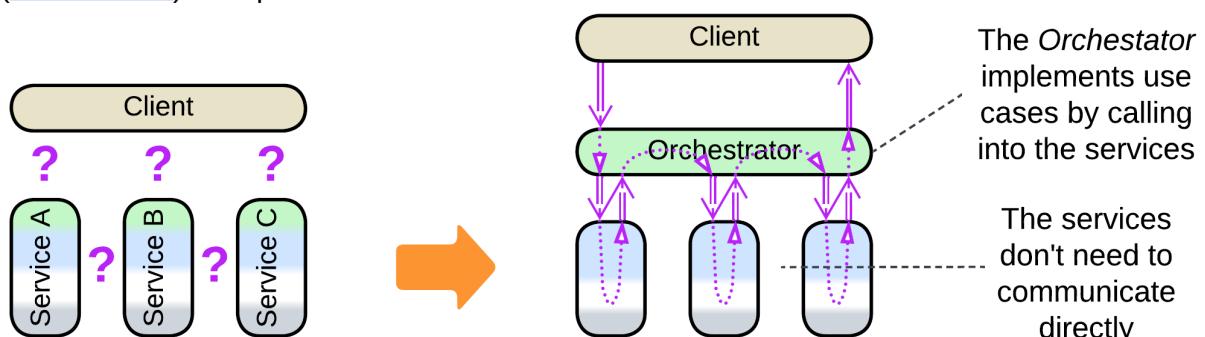
We have deconstructed the most common programming paradigms into their driving forces and shown how those forces shape distributed architectures:

- An object-oriented system relies on hierarchical decomposition of a complex domain, just like *SOA* and *Orchestrated (Micro-)Services* do.
- Functional programming streams data through a sequence of transformations, which is the idea behind *Choreographed Event-Driven Architecture* and *Data Mesh*.
- Procedural style lets any piece of logic access the entire project's data, resembling *Space-Based Architecture* and *Services with a Shared Database*.

Now let's examine each of these approaches in depth:

Orchestration

The most straightforward way to integrate services is to add a coordinating layer ([Orchestrator](#)) on top of them:

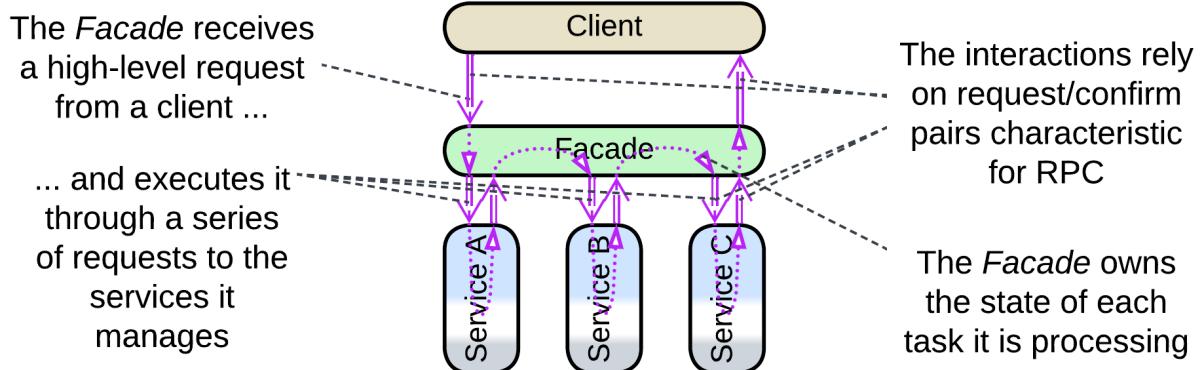


The good thing is that your *Orchestrator* has explicit code for every use case it covers and every running scenario gets an associated thread, coroutine, or object so that you are able to attach to the *Orchestrator* and debug any use case step by step. Nor do you have to worry about keeping the state of the services consistent as they are passive with all the changes in the system being driven by the *Orchestrator*.

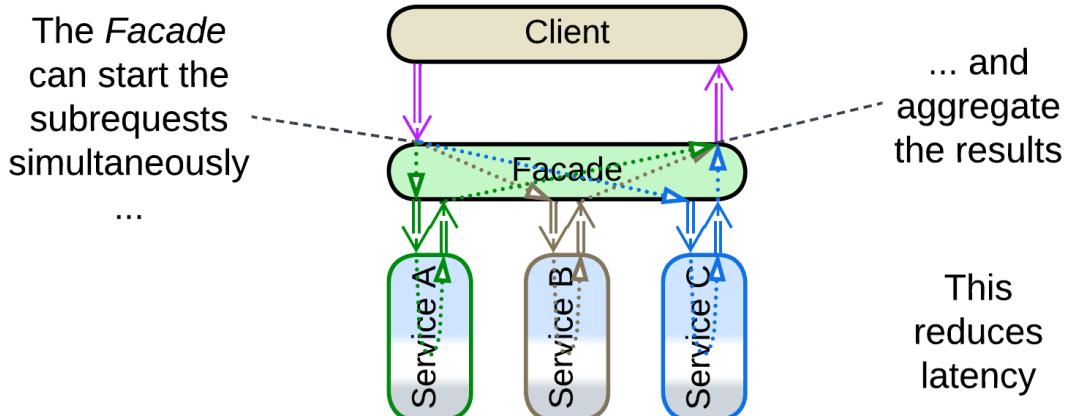
Orchestration is the default approach for single-process (desktop) applications where it is faster to call into an orchestrated module and return than to send it a message. However, in distributed systems orchestration doubles the communication overhead (when compared to [choreography](#) or [shared data](#)) as every method call into an orchestrated service uses two messages: request and confirmation.

Roles

In a backend which serves client requests an *Orchestrator* takes the role of *Facade* [[GoF](#)] – a module that provides and implements a high-level interface for a multicomponent system. It sends requests to the underlying services and waits for their confirmations – the mode of action that can be wrapped in an [RPC](#) (*remote procedure call*). The state of each scenario that the facade runs resides in the associated thread's or coroutine's call stack (for [Reactor](#) [[POSA2](#)] or [Half-Sync/Half-Async](#) [[POSA2](#)] implementations, correspondingly) or in a dedicated object (for [Proactor](#) [[POSA2](#)]).

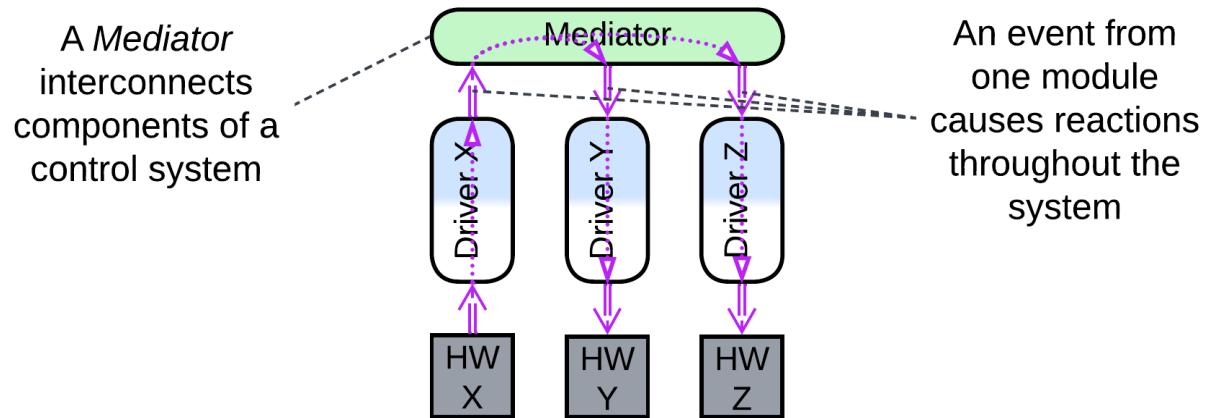


A *Facade* also supports querying the services in parallel and collecting the data returned into a single message through the *Splitter* and *Aggregator* patterns of [[EIP](#)]. That reduces latency (and resource consumption as the whole task is completed faster) for [scatter or gather](#) requests when compared to sequential execution.



Embedded and system programming – the areas that deal with automating [control](#) of hardware or distributed software – employ *Orchestrators* as *Mediators* [[GoF](#)] – components that keep the state of the whole system (and, by implication, any hardware it may manage) consistent by enacting a system-wide reaction to any observable change in any of the system's constituents. A mediator operates in non-blocking, fire-and-forget mode which is more characteristic of choreography, to be discussed [below](#). This also means that you will

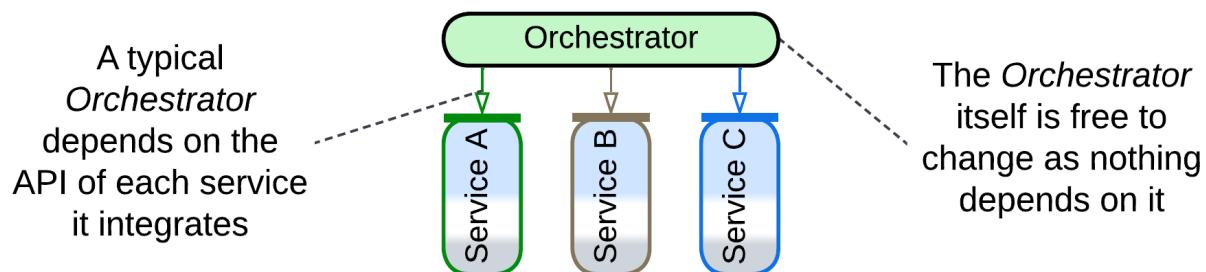
not be able to debug a use case as a thread – because [there are no predefined scenarios in control software!](#)



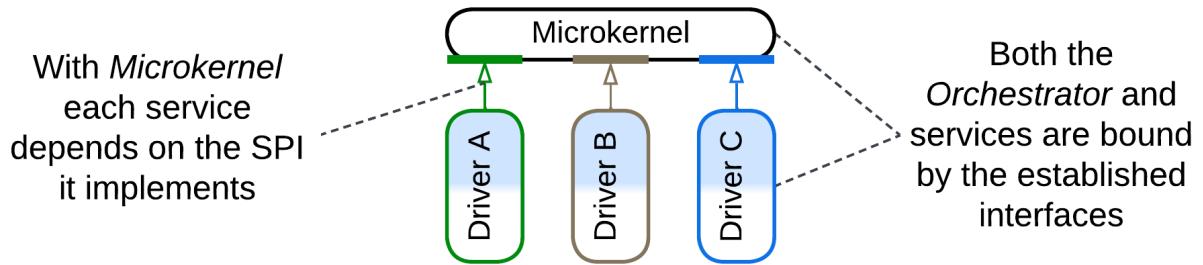
Such a difference may be rooted in the direction of the control and information flow: in a backend it comes as a high-level command while control systems react to low-level events.

Dependencies

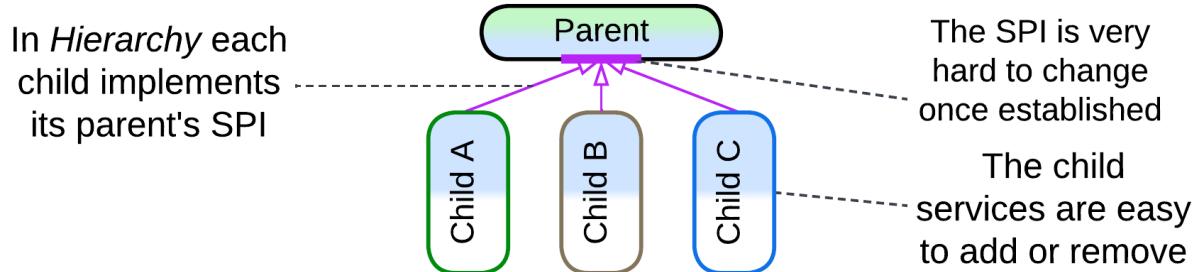
By default an *Orchestrator* depends on each service which it manages – that means that a change in a service's interface or contract – caused by fixing a bug, adding a feature, or optimizing performance – requires corresponding changes in the *Orchestrator*. That is acceptable as the *Orchestrator*'s client-facing, high-level logic tends to evolve much faster than the business rules of the lower layer of services, therefore the team behind the orchestrator, unrestricted by other components depending on it, will likely release way more often than any other team. However, as the number of the managed services and the lengths of their APIs increase, so does the amount of information that the *Orchestrator*'s team must remember and the influx of changes they must integrate in their code. For a large project the workload of supporting the *orchestration layer* may paralyze its development – that was a major reason behind the decline of [Enterprise SOA \[FSA\]](#) where [ESB](#) used to orchestrate all the interactions in the system, including those between domain-level services and components of the utility layer.



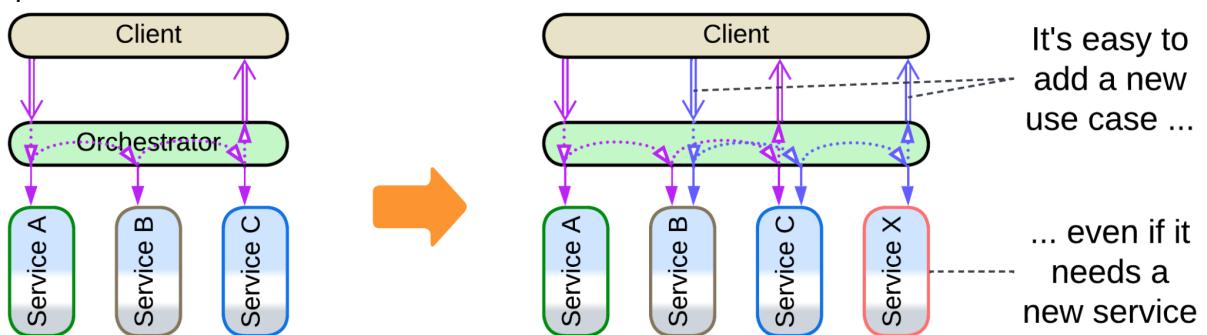
Another option, which appears in [Plugins](#) and develops in [Microkernel](#) and [Hexagonal Architecture](#) stems from [dependency inversion](#): the *Orchestrator* defines an [SPI](#) (service provider interface) for every service. That makes each service depend on the *Orchestrator* so that a single *Orchestrator*'s team does not need to follow updates of the multiple services' APIs – instead it initiates the changes at its own pace. However, with that approach the design of an SPI requires coordination from the teams on both sides of it and the once settled interface becomes hard to change. The most famous example of modules that implement SPIs are OS drivers.



Furthermore, some domains develop that idea into a *Hierarchy*: when services implement related concepts, they may match a single SPI, making the *Orchestrator* simpler (as there is no more need to remember multiple interfaces). That is the case with telecom or payment gateways and it may also be found with trees of product categories in online marketplaces.

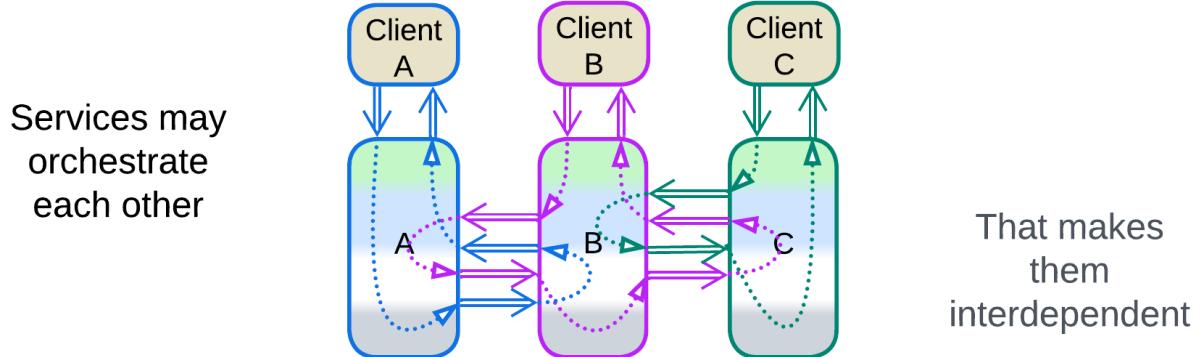


All kinds of orchestration allow for an easy addition of new use cases which may even involve new services as that changes nothing in the existing code. However, removing or restructuring (splitting or merging) previously integrated services requires much work within the orchestrator, except for in a *Hierarchy* where all the services implement the same interface which means that the code in the *Orchestrator* does not depend (much) on any specific child.

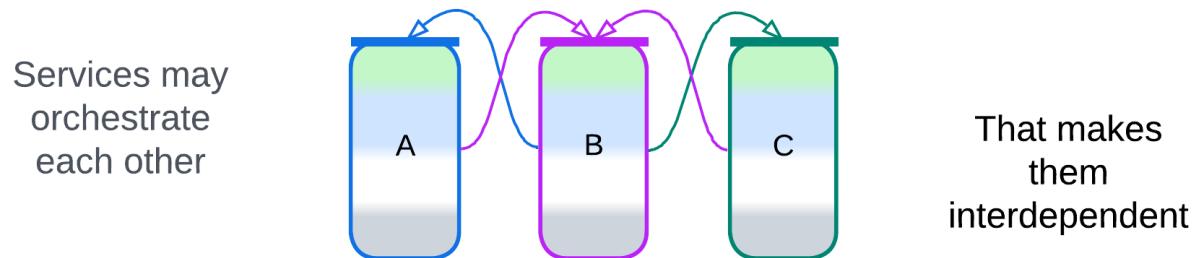


Mutual orchestration

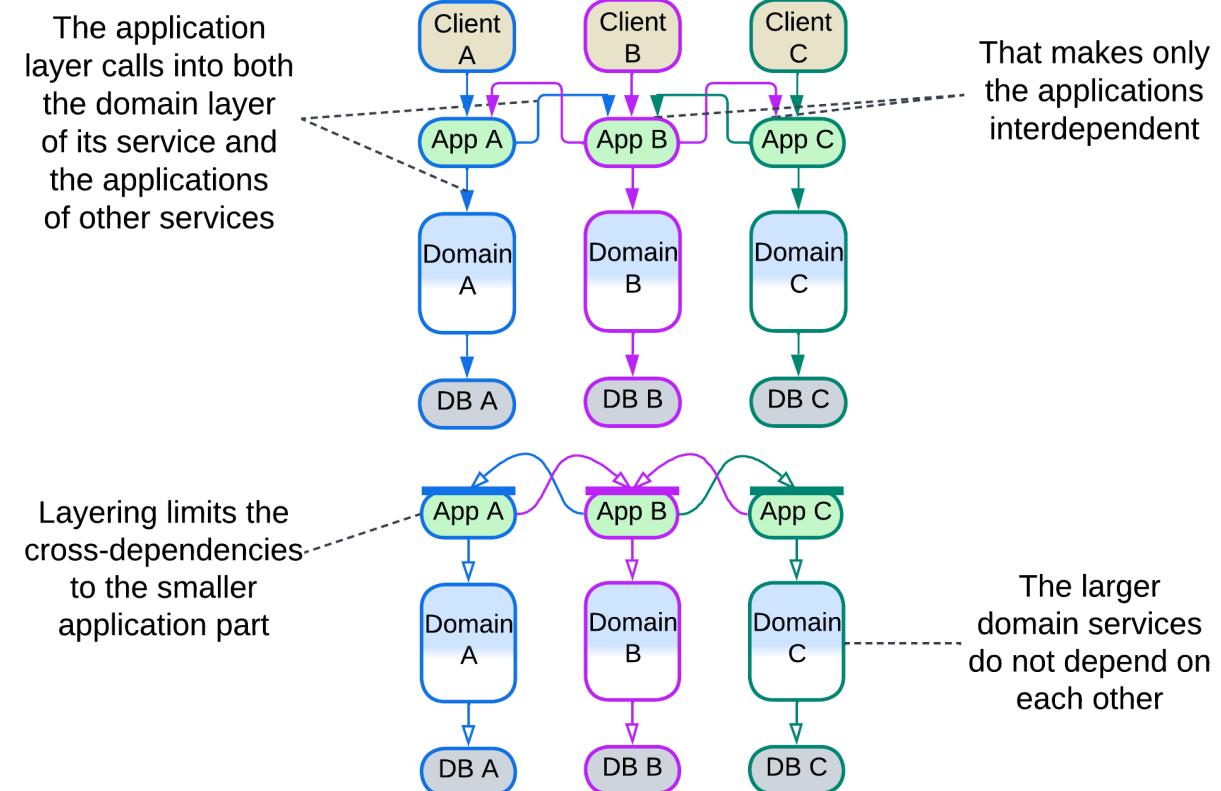
In some systems there are several services that have their own kinds of clients (for example, employees of different departments). Each of the services tries hard to process its clients' requests on its own but occasionally still needs help from other parts of the system. This creates a paradoxical case where several services orchestrate each other:



As each of the services depends on the APIs of the others, any change to any interface or composition of such a system requires consent and collaboration from every team as it impacts the code of all the services.



In real life services are likely to be layered, with their upper layers acting as both internal and external *Orchestrators*. Layering isolates interdependencies to the relatively small application-level components and resolves, to an extent, the seemingly counterintuitive case of mutual orchestration as now there is an explicit, though fragmented, system-wide orchestration layer.

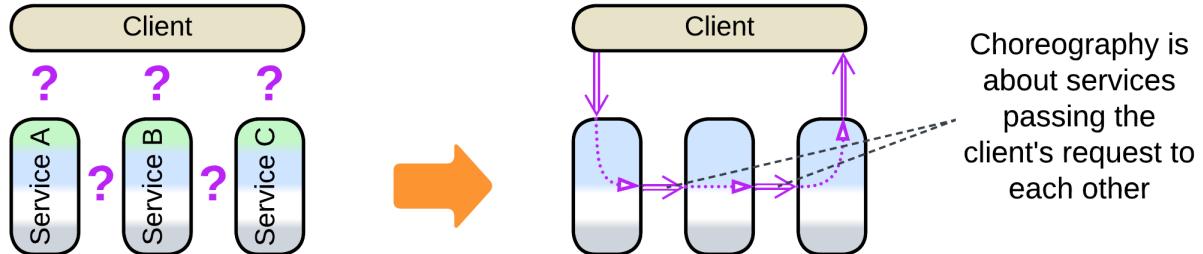


Summary

Orchestration represents use cases as a code, allowing for an orchestrated system to support many complex scenarios. Dealing with errors is as trivial as properly handling exceptions. This approach trades performance for clarity.

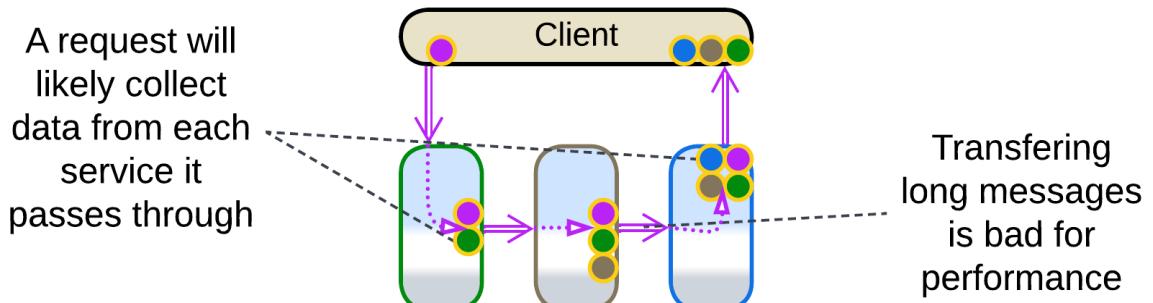
Choreography

Another integration option is to build a [Pipeline](#) to pass every client's request through a chain of components:

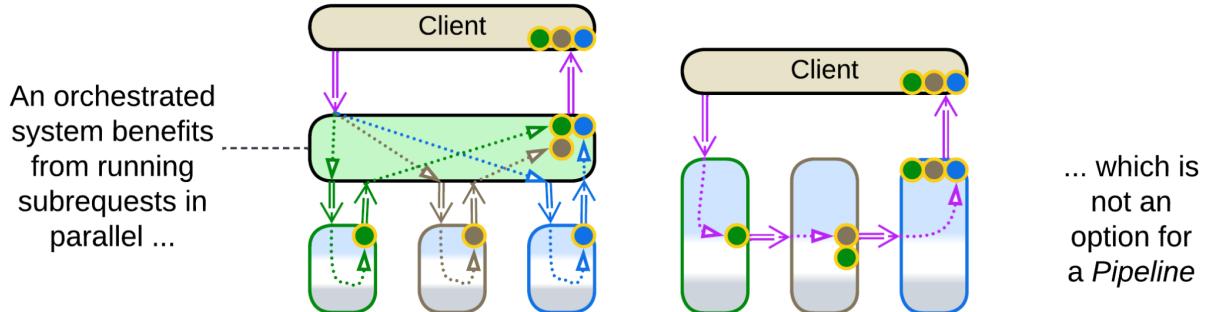


In that case there is no owner for *workflows* – each request is just a data packet which is transformed multiple times as it passes through the *Pipeline*. Debugging is mostly limited to reading logs as there is no dedicated component to connect to for single-step execution of a use case. Nor is there a single piece of code to define each of the system-wide scenarios – their logic emerges from the graph of event channels that connect services and from messages that each involved event handler sends. Consistency of the services' states is the responsibility of the services themselves as there is none supervising them.

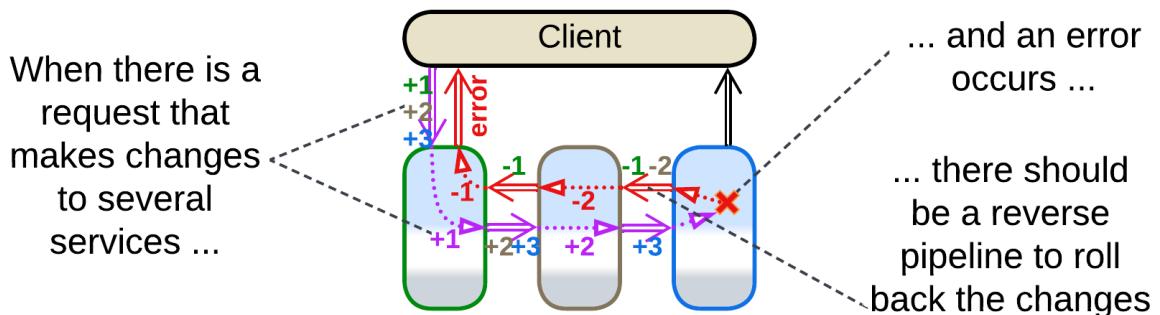
On the bright side, there is no communication overhead caused by response messages as there are no responses – the processing cost is one message per service, half of the cost for an orchestrated architecture. Still, messages in choreographed systems tend to be longer than those used with [orchestration](#) as each message needs to carry the entire request's state – there is no [Orchestrator](#) to own the state and distribute parts of the payload among involved services.



Latency may also be suboptimal as parallelizing execution of a request is easier said than done because there is no place (*Aggregator* [[EIP](#)]) to collect multiple related messages, which also means that there is no associated cost in resources (RAM and CPU time) for storing their fragments. Please note that an *Aggregator*, when added, starts orchestrating the system – it stands between the client and services and meddles with the traffic and logic. It spends resources to store the received messages for aggregation, and the messages start forming request/confirm pairs – which are characteristic of orchestration.

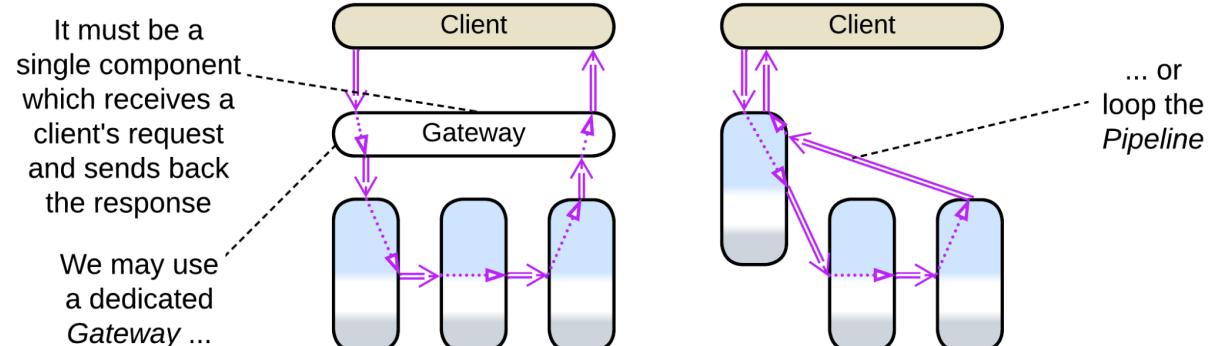


Still another trouble with choreography comes from its weakness in error processing. When a service in the middle of a request processing pipeline encounters an error, it cannot generate its normal output to be sent further downstream. One option is to fill in a null (or error) value but then each receiver of the message should remember to check for null and know how to deal with an error. Another way is adding a dedicated error channel for each service to push failed requests into, but that complicates the whole system's structure. Moreover, a failure in the middle of processing a request may cause the services to end up with inconsistent data if no special attention (a new kind of request to compensate the original one) is paid to roll back the partial change. Please note that all of that is conveniently handled by an *Orchestrator*.

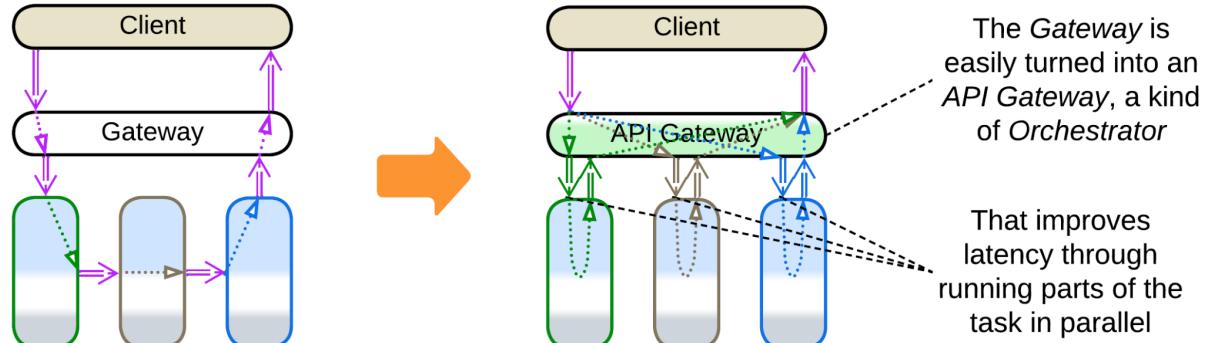


Early response

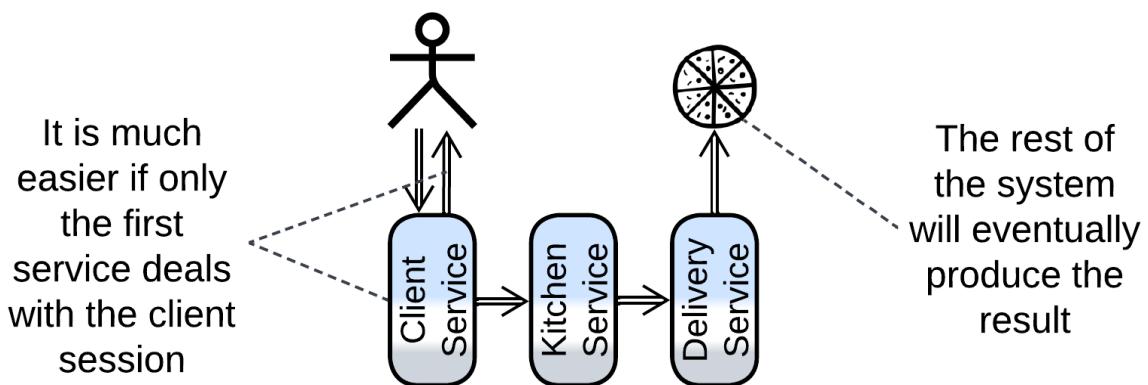
The ordinary mode of action for a pipeline – sending the final results of processing to the client – requires either for the tail of the pipeline to send data to its head or for existence of a stateful intermediate component – [Gateway](#) – to receive the client's request, forward it to the head of the pipeline, wait on the pipeline's tail for the result of processing, and return it to the client. That is necessary because a client would usually open a single connection which is impossible to share between multiple services, namely the (receiving) head and (sending) tail of the pipeline.



The gateway, if used, may parallelize processing of [scatter or gather](#) requests by turning into an [API Gateway](#) which is a kind of *Orchestrator*. Which means that the system changes its paradigm from choreography to orchestration.



It is possible to avoid both adding a *Gateway* and having the cyclic dependency if clients don't immediately need the final results of processing their requests. In such a case the service which receives the original request does its (first) step of processing, sends the response to the client, and then notifies services down the pipeline. Though such a use case seems to be unlikely, it happens in real life, for example, with pizza delivery. As soon as a buyer fills in their contact details and pays for the food, the order can be confirmed and forwarded to the kitchen. When it is ready it's forwarded to the delivery, and finally the physical goods appear at the buyer's door.

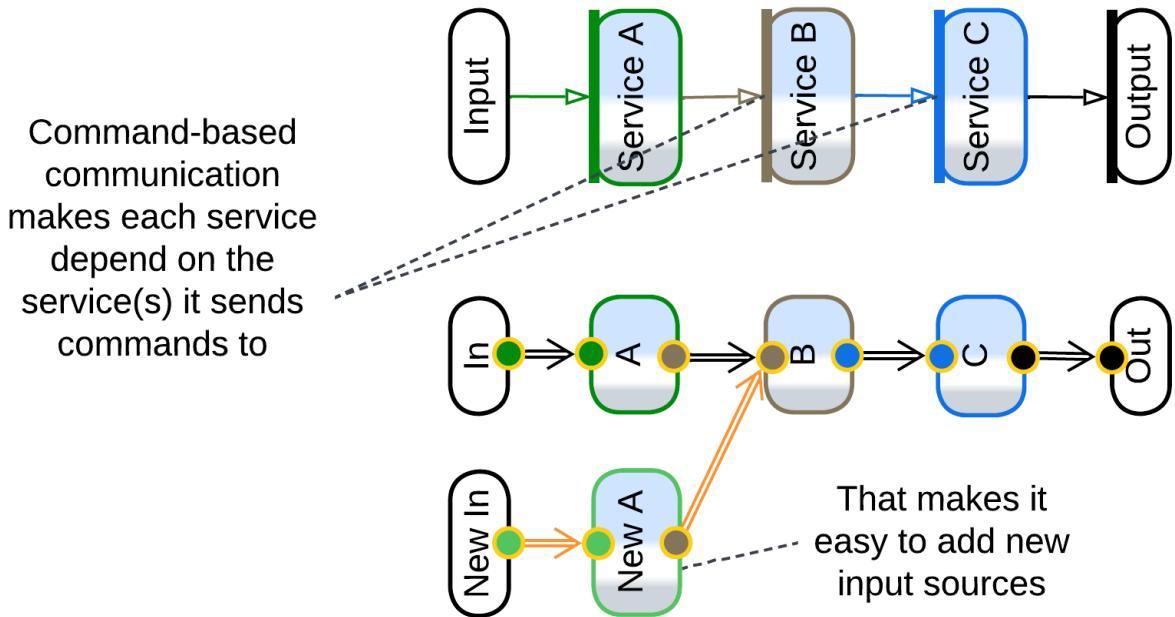


Early response allows for choreography to shine in its purest form: with extensibility, high performance, but also high latency. A similar approach may be used in [Service-Based Architecture \[FSA\]](#) (aka Macroservices) [for communication between the services](#) (*bounded contexts*) if they only need to notify each other of events without waiting for responses.

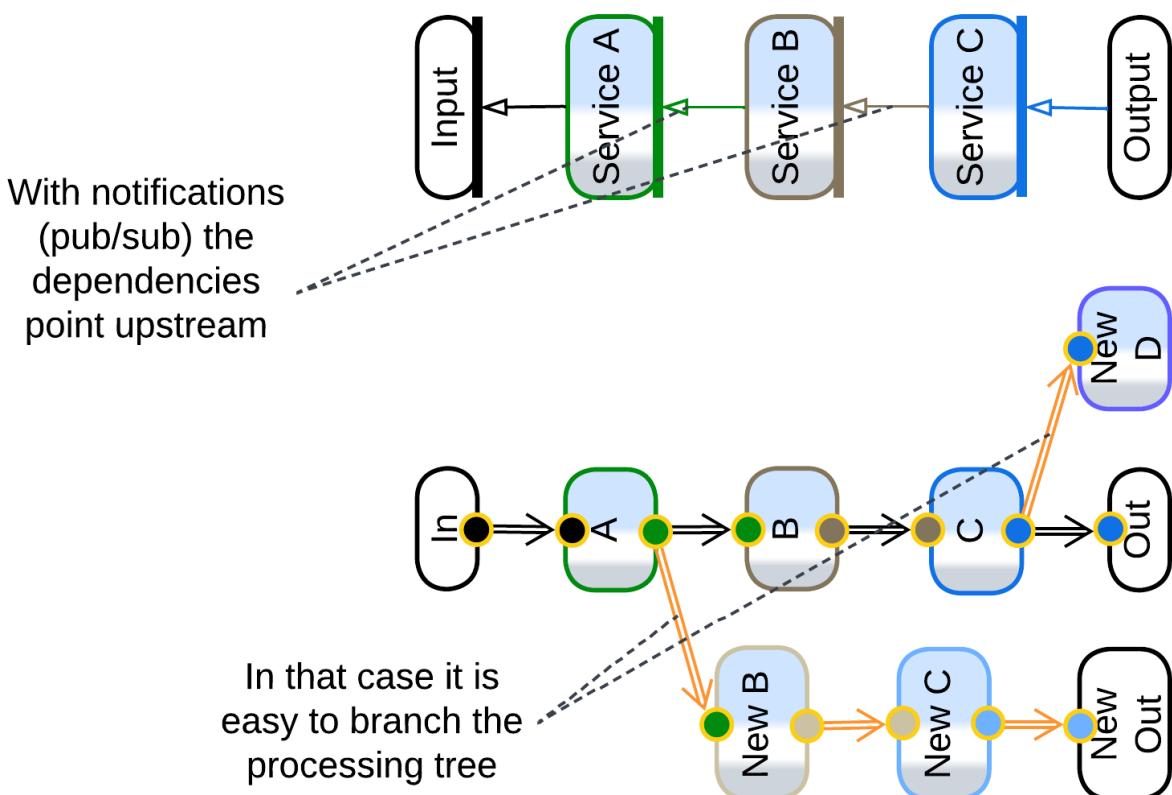
Dependencies

A pipeline may be built with downstream or upstream dependencies or with a shared schema.

If services communicate through commands, each service depends on all the direct destinations of its commands as it must know each of the APIs which it uses. This mode of communication is mostly used with [Actors](#) that power embedded, telecom, messengers, and some banking systems. Downstream dependencies make it easy to add input chains (upstream services that deal with new hardware or external components) although changing anything at the output end of the pipeline is going to break the input parts that send messages to the component changed.

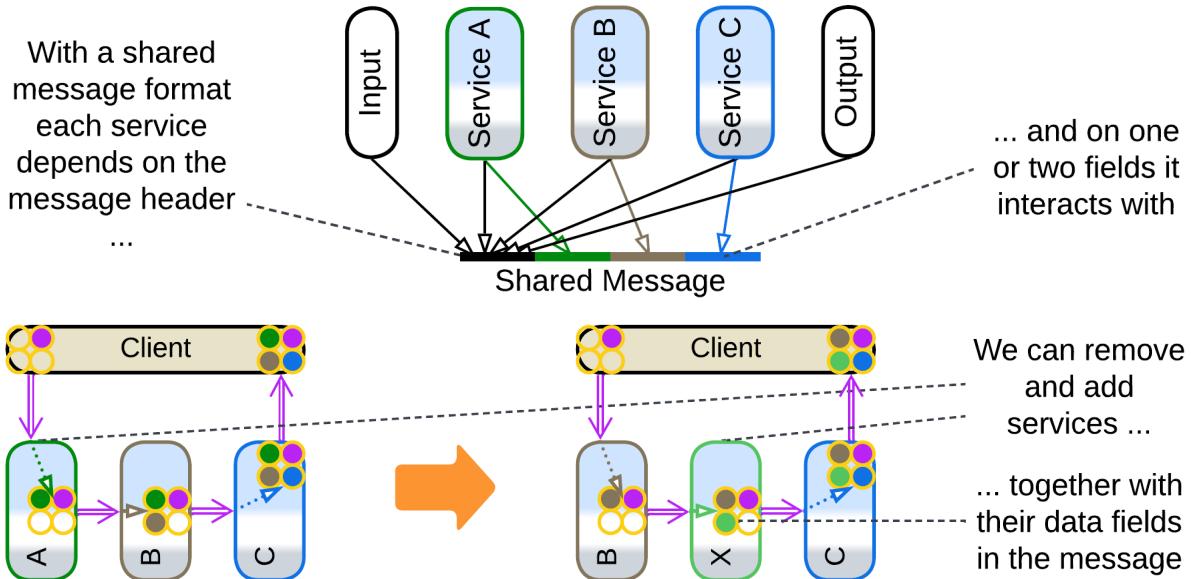


Upstream dependencies come from the [publish/subscribe](#) model where each service broadcasts notifications about what it has done to any interested subscriber. This way of building systems engines [Event-Driven Architecture \[FSA\]](#) which is used in high-load backends. Extending or truncating an already implemented request processing tree is as easy as adding or removing subscribers to existing events but the creation of a new event source will require changes in the downstream components. The easy addition of downstream branches supports new customer experiences and analytical features which business is hungry for.



The final option is for the entire pipeline to use a uniform message format ([Stamp Coupling \[SAHP\]](#)) which often contains one dedicated field per service involved. This way a

service depends only on the message header (with the list of the fields and a record id) and the format of the single field it reads (stores data) or writes (retrieves data as *Content Enricher* [EIP]). That works well with system-wide queries but binds all the services to the schema of the message in a way similar to accessing a shared database (to be discussed [below](#)). Such an architecture decouples the services to the extent that any of them can be freely added or removed, together with the message field(s) it fills or reads.

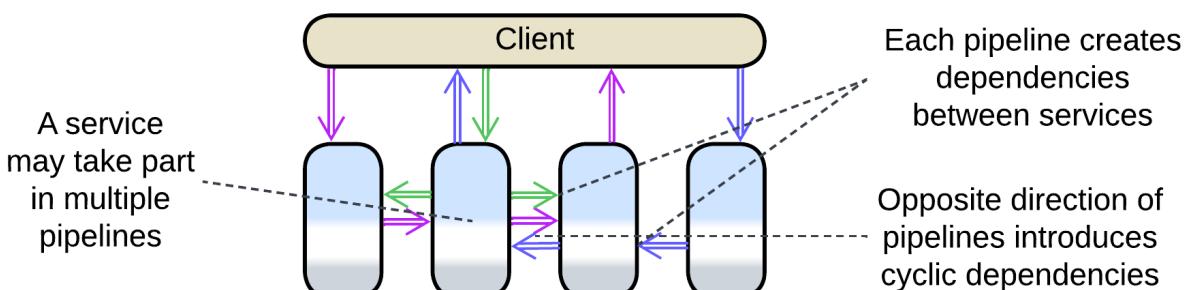


A peculiar feature of choreography is the ability to cut and cross-link pipelines with compatible interfaces by changing a single service (or even system configuration if you build it with communication channels). That gives it a lot of flexibility – as long as you can comprehend all the dependencies (and channels) in the system, which becomes non-trivial as it grows.



Multi-choreography

It is very common for a service to participate in multiple pipelines, especially if it owns a database – as there should be a use case which fills in the data and at least one other scenario which reads from that database. Each pipeline makes the service depend on one or more interfaces it communicates with, which often belong to multiple services, coupling components of the system and making it impair future structural changes.



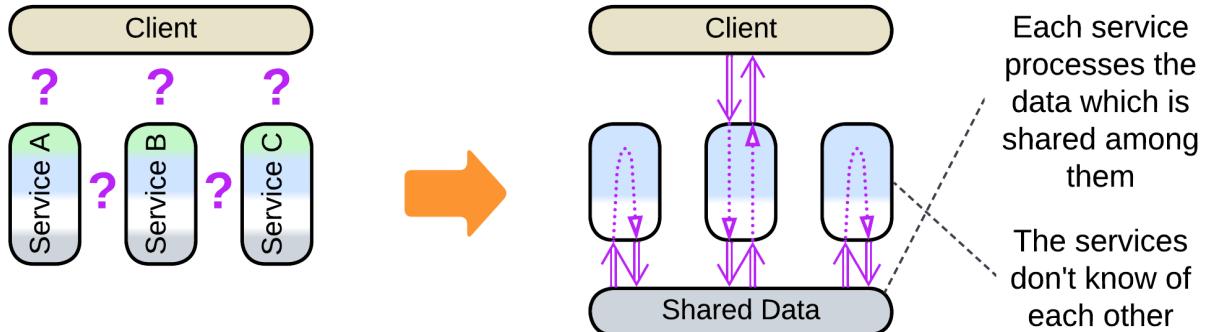
Summary

Overall, choreography seems to be a lightweight approach that prioritizes throughput over latency and is suitable for highly-loaded scenarios of limited complexity. However, a choreographed system will likely become unintelligible if it is made to support more than a few use cases.

There is a decent [overview from Microsoft](#).

Shared data

The final approach is integration through shared data ([Shared Repository](#)):

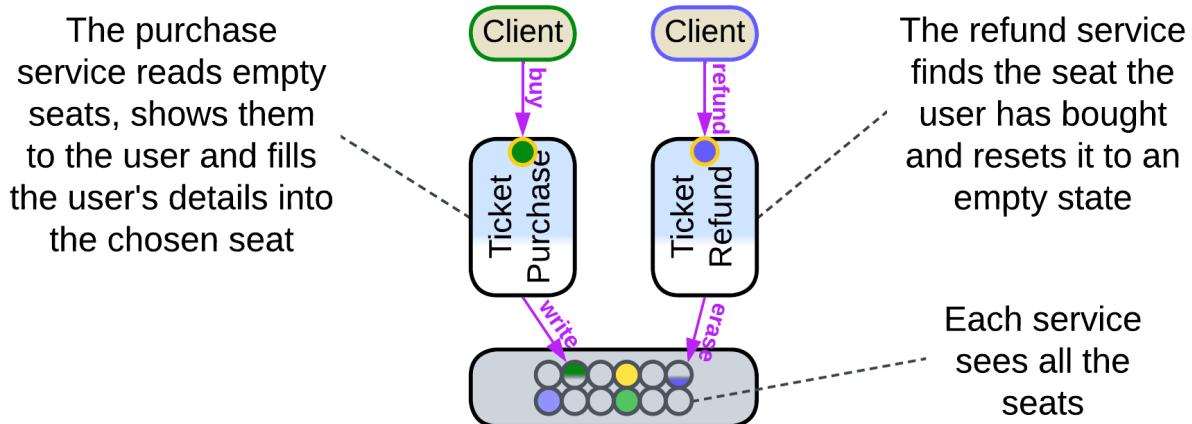


The shared data is a “blackboard” available for each service to read from and write to. It is passive (as controlled by the services) and does not contain any logic except for the data schema, which represents a part of the domain knowledge. That makes communication through shared data the antipode of [orchestration](#), which also features a shared component, [Orchestrator](#), which is, however, active (controls services) and contains business logic, not data.

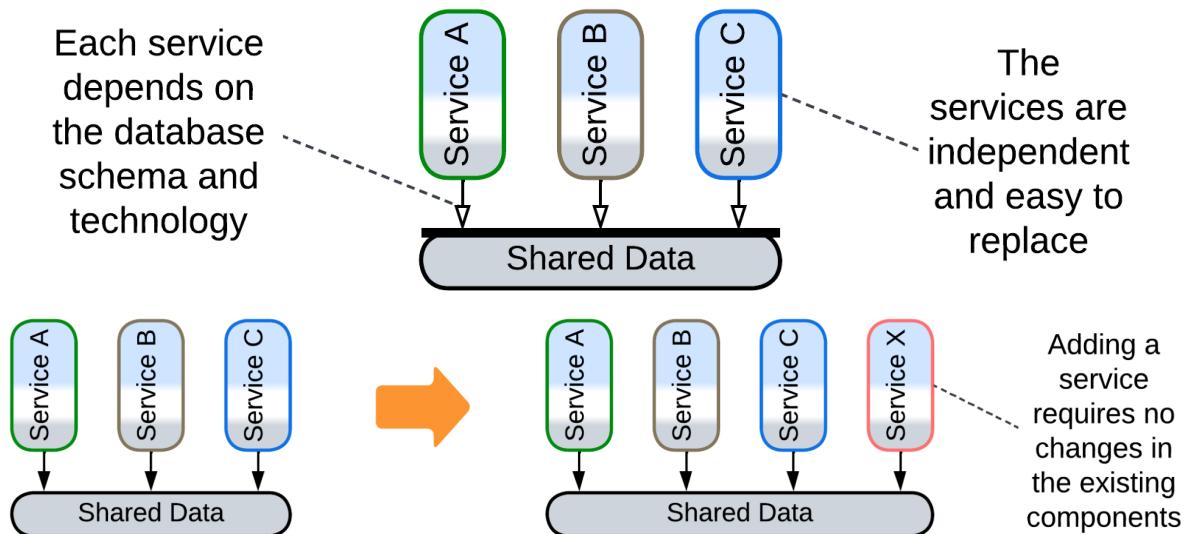
Shared data can be used for storage, messaging, or both:

Storage

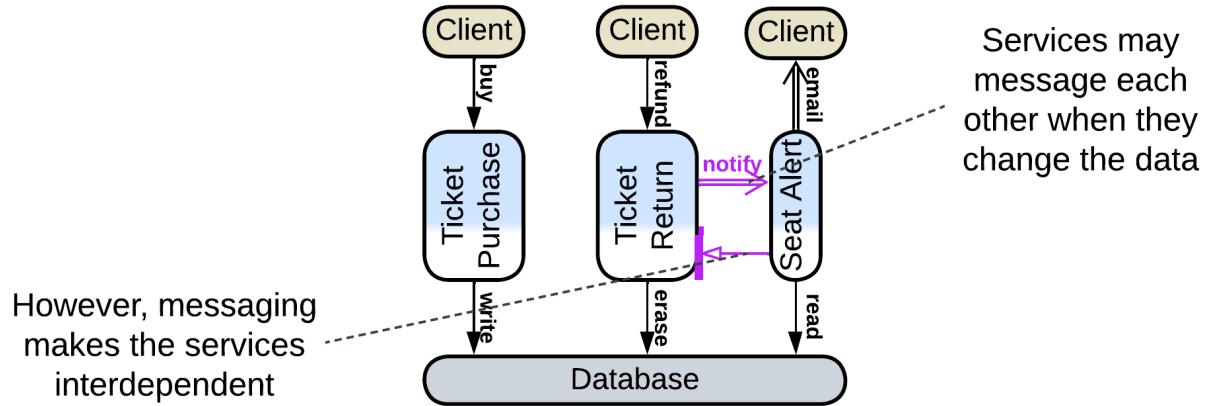
The most common case of shared data is storage (usually a database, sometimes a file system) for a (sub)domain that has functionally independent services which operate on a common dataset. For example, a ticket purchase service and a ticket refund service share a database of ticket details. The ticket purchase service reads in the available seats and fills in ticket data for purchases. The ticket refund service should be able to find all tickets bought by a user and delete the user data from seats refunded. The only communication between the purchase and refund services is the shared database of tickets or seats, so that one of them sees the changes made by the other the next time it reads the data.



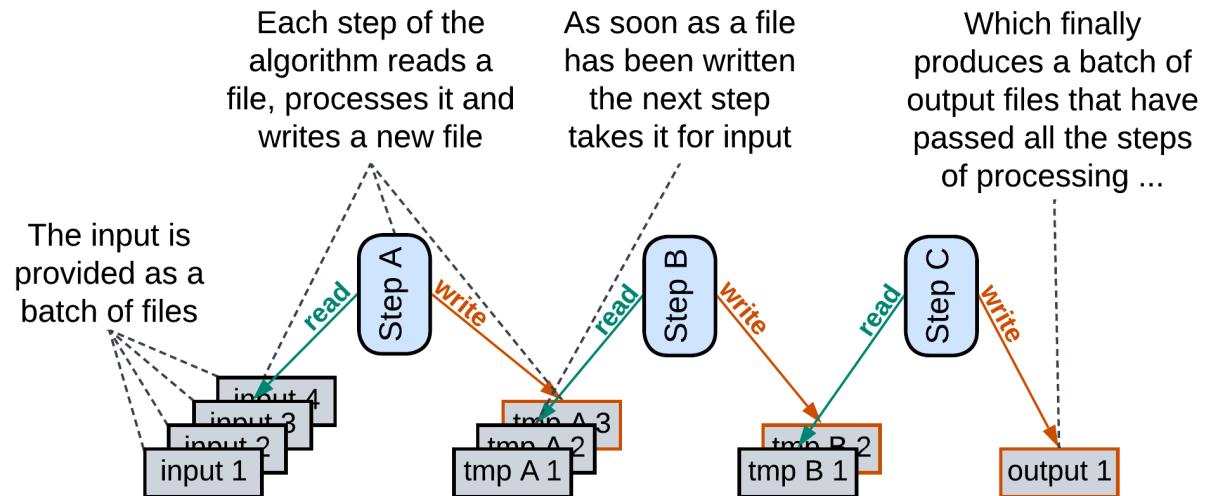
With this model the services don't depend on each other – instead, they depend on the shared (domain) data format and the database technology. Thus, it is easy to add, modify, or remove services but hard to change the shared data structure or, especially, the database vendor.



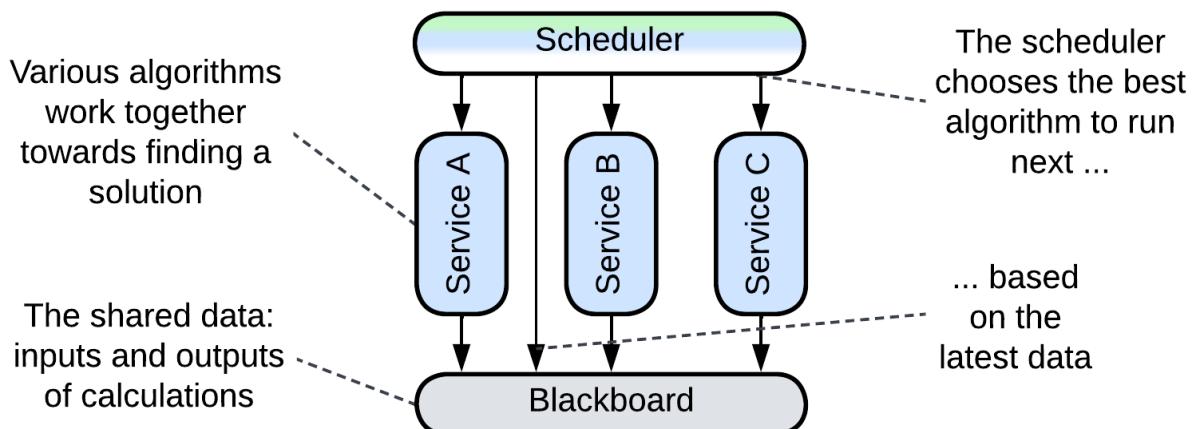
Services usually need to coordinate their actions. Commonly, services with a shared database rely on a messaging [Middleware](#) for communication. Users of our ticketing system will want to be notified (through email, SMS or an instant message) when a free seat that they are interested in appears. We're not going to complicate either of the existing services by integration with instant messengers, so we will create a new notification service, which must track each returned ticket to see if any user wants to buy it. This is easily implemented by the return service publishing and the notification service subscribing to a ticket return event, mixing in a bit of choreography into our data-centric backend.



Another case is found with data processing pipelines where an element may periodically read new files from a folder or new records from a database table to avoid implementing notifications. This increases latency and may cause a little CPU load when the system is idle, but is perfectly ok for long-running calculations.

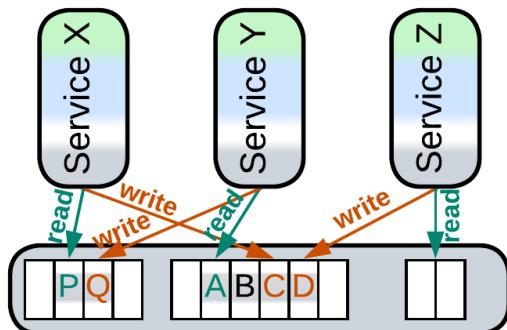


Finally, there is the rarely used option of an external [Scheduler](#) which selects the services which should run based on the data available. This is known as the [Blackboard pattern](#) and [something similar](#) happens in 3D game engines. The *Scheduler* (which is an [Orchestrator](#), by the way) is needed when CPU (or GPU or RAM) resources are much lower than what the services would consume if all of them ran in parallel, thus they must be given priorities, and the priorities change based on the context which is regularly estimated from the system's data.

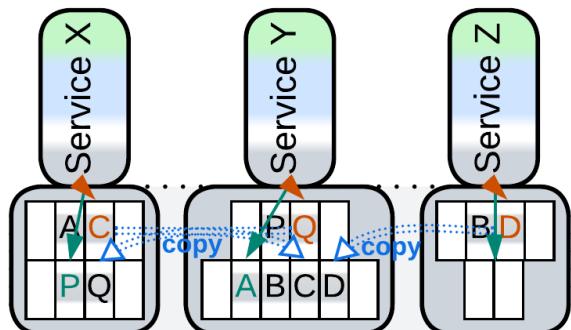


Messaging

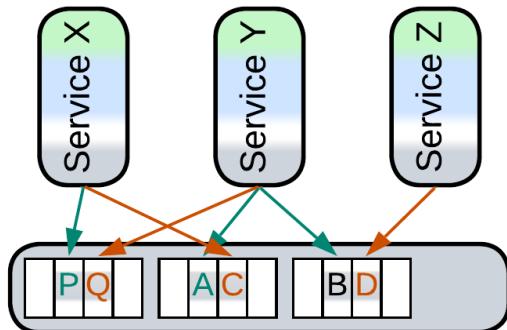
The other, not as obvious, use case for shared data is messaging, which is implemented by the sender writing to a (shared) queue (or log) while the recipient is waiting to read from it. Queues can be used for any kind of messages: request/confirm pairs, commands, or notifications. Each service may have a dedicated queue (either input for commands mode or output for notifications), a pair of queues (messages from the service's output are duplicated by an underlying distributed [Middleware](#) to input queues of their destinations), or there may be a queue per communication channel, or a single queue for the entire system (or a global queue per message priority) with each message carrying destination id (for commands) or topic (for notifications).



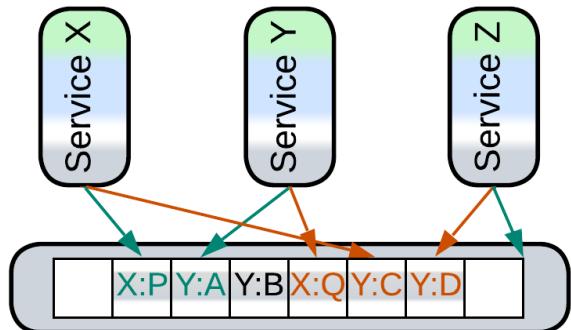
A Queue per Service



Input and Output Queues



A Queue per Channel



A Single System Queue

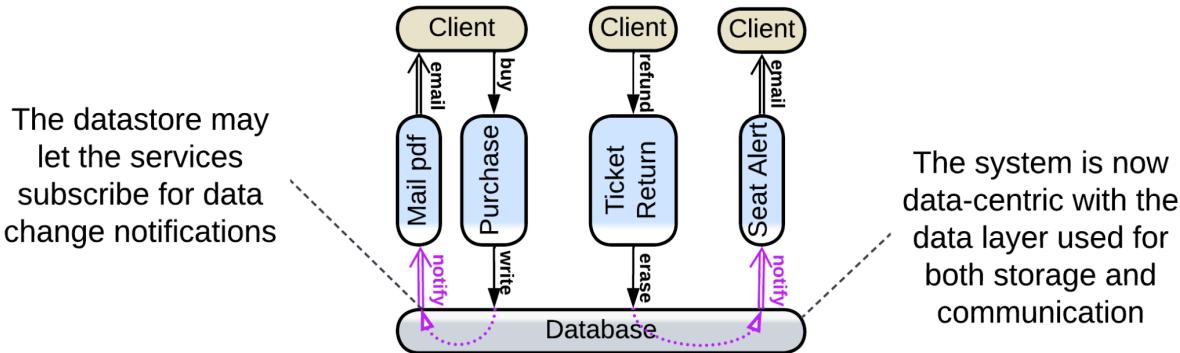
The use of shared data for messaging turns our datastore into a [Middleware](#). The dependencies are identical to [those in choreography](#) – each service depends on the APIs of its destinations for commands or its sources for notifications.

There should be a means for the recipient of a message to know about its arrival so that it starts processing the input. Usually a messaging *Middleware* implements a `receive()` method for the service to block on. However, very low latency applications, like [HFT](#), may [busy-wait](#) by repeatedly re-reading the shared memory so that the service starts processing the incoming data immediately on its arrival, bypassing the OS scheduler. This is the fastest means of communication available in software.

Full-featured

Finally, some (usually distributed) datastores implement data change notifications. That allows for the services to communicate through the datastore in real-time, removing both the need for an additional *Middleware* and interdependencies for the services. Such a system

follows the [Shared Repository](#) pattern of [POSA4] which was rectified as [Space-Based Architecture](#) [SAP, FSA]. In our example, the available seats notification service subscribes to changes in the seats data in the database – this way it does not need to be aware of the existence of other services at all. We can also move the email notifications logic of the ticket purchase service into a separate component which would track purchases in the database and send a printable version of each newly acquired ticket to the buyer's email address which can be found in the ticket details in the database.



Summary

Communication through shared data is best suited for data-centric domains (for example, ticket purchase). It allows for the services to be unaware of each other's existence, just as they are with orchestration, but the structure of the domain data becomes hard to change as it is referenced all over the code. Shared data may also be used to implement messaging.

Comparison of the options

We have briefly discussed three approaches to communication: orchestration, choreography, and shared data. Let's recall when it makes sense to use each of them.

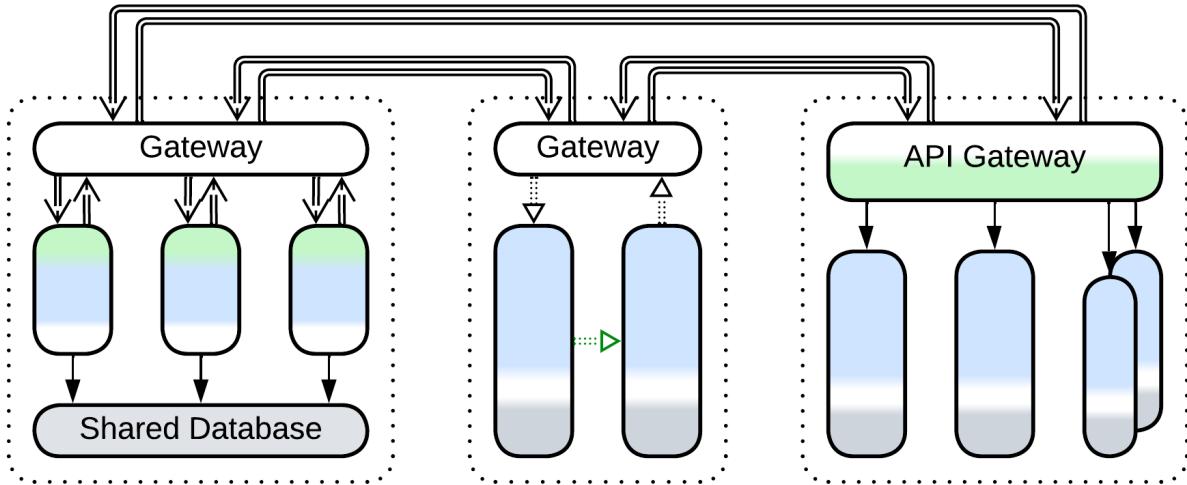
[Orchestration](#) is built around use cases. They are easy to program and add, no matter how complex they become. Thus, if your (sub)domain is coupled, or your understanding of it is still evolving, this is the way to go, as you will be able to change the high-level logic in any imaginable way because you express it as convenient imperative code.

[Shared data](#) is all about... er... domain data. If you really (believe that you) know your domain, and it deals with coupled data – this is your chance. You may even add in an *Orchestrator* if there are use cases that involve multiple subdomains. The business logic is going to be easy to extend while changes to the data schema are sure to cause havoc.

[Choreography](#) pays off for weakly coupled domains with a few simple use cases. It has good performance and flexibility, but lacks the expressive power of orchestration and becomes very messy as the number of tasks and components grows. It works best with independent teams and delayed processing – when users do not wait for the immediate results of their actions.

There is advice [from Microsoft](#) and [DEDS] which makes perfect sense: use choreography for communication between *bounded contexts* (subdomains) but revert to orchestration (or maybe shared data) inside each context. Indeed, subdomains are likely to be loosely coupled while most user requests don't traverse subdomain boundaries – which kindles hope that their interactions are few and not time-critical. If we follow the advice, we get [Cell-Based Architecture](#) ([WSO2 definition](#)), which collects the best of two worlds:

orchestration and/or shared data for strongly coupled parts and choreography between them.



By the way, you could have noticed a few odd cases:

- An [Orchestrator](#) in a [control system](#) does not run scenarios and its mode of action resembles choreography.
- A choreographed system may use a [shared message format](#), which makes it resemble a system with shared data, even though no shared database is present.
- A shared database may be used to [implement messaging](#) for an orchestrated or choreographed system, effectively becoming a [Middleware](#).

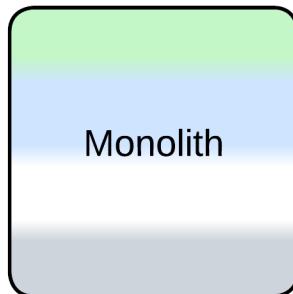
That likely means that our distinction between the modes of communication is a bit artificial and there may be a yet unknown deeper model to look for.

Part 2. Basic Metapatterns

Basic metapatterns are both common stand-alone architectures and building blocks for more complex systems. They include the single-component *Monolithic* architecture and the results of its division along each of the [coordinate axes](#), namely *abstractness*, *subdomain*, and *sharding*:

Monolith

A *Monolith* is a (sub)system the internals of which we prefer not to see



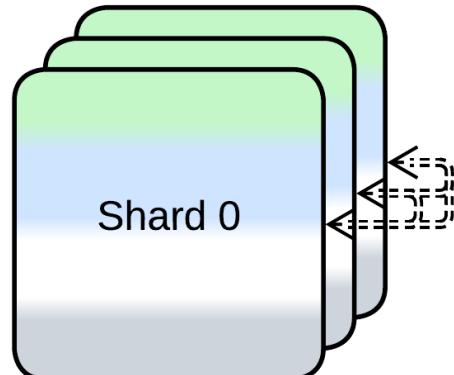
It may lack an internal structure or its components may be coupled

Monolith is a single-component system, the simplest possible architecture. It is easy to write but hard to evolve and maintain.

Includes: Reactor, Proactor, and Half-Sync/Half-Async.

Shards

Shards are multiple instances of a subsystem



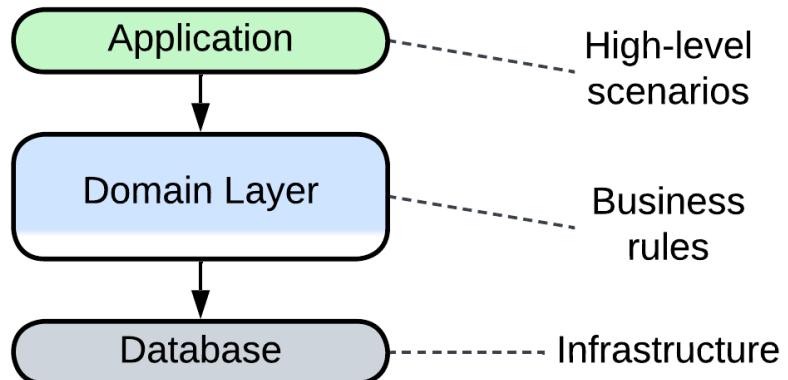
A shard may be stateless or stateful

Shards are multiple instances of a *Monolith*. They are scalable but usually require an external component for coordination.

Includes: Shards and Amazon Cells, Replicas, Pool of Stateless Instances, and Create on Demand.

Layers

Layers divide the system by the level of abstractness

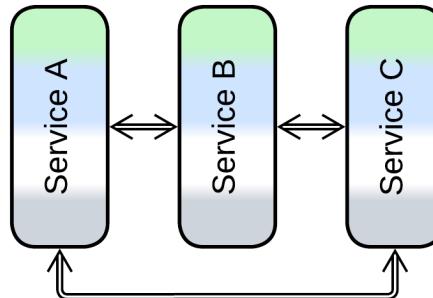


Layers contain one component per level of abstraction. The layers may vary in technologies and forces and scale individually.

Includes: Layers and Tiers.

Services

Each service implements a subdomain



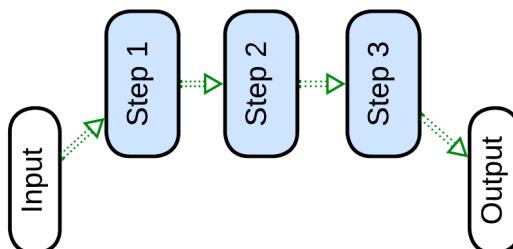
The subdomains should be loosely coupled

Services divide a system into subdomains, often resulting in parts of comparable size assignable to dedicated teams. However, a system of Services is hard to synchronize or debug.

Includes: Service-Based Architecture, Modular Monolith (Modulith), Microservices, Device Drivers, and Actors.

Pipeline

The system processes data in a sequence of steps



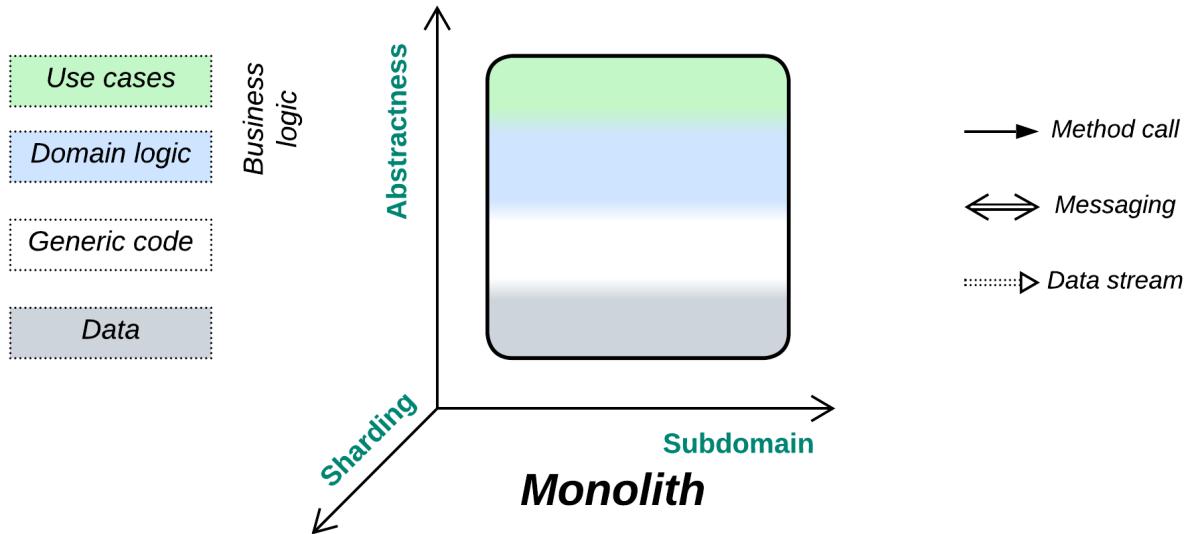
Each step knows nothing of its neighbors

A Pipeline is a kind of Services with unidirectional flow. Each service implements a single step of data processing. The system is flexible but may grow out of control.

Includes: Pipes and Filters, Choreographed Event-Driven Architecture, and Data Mesh.

Monolith

Let's take a look at the simplest possible [metapattern](#) – *Monolith* – and see what it can teach us.



Keep it simple, stupid! If you don't need a modular design, why bother?

Known as: Monolith.

Variants:

By internal structure:

- True Monolith / [Big Ball of Mud](#),
- (misapplied) Layered Monolith [[FSA](#)],
- (misapplied) Modular Monolith [[FSA](#)] (Modulith),
- (inexact) Plugins [[FSA](#)] and Hexagonal Architecture.

By mode of action:

- [Reactor](#) [[POSA2](#)],
- [Proactor](#) [[POSA2](#)],
- (inexact) [Half-Sync/Half-Async](#) [[POSA2](#)],
- (inexact) [\(Re\)Actor-with-Extractors](#).

Structure: A monoblock with no strong internal modularity.

Type: Main, root of the hierarchy of metapatterns.

Benefits	Drawbacks
Rapid start of development	Quickly deteriorates with project growth
Easy debugging	Hard to develop with multiple teams
Best latency	Does not scale
Low resource consumption	Lacks support for conflicting forces
The system's state is self-consistent	Any failure crashes the entire system

References: [Big Ball of Mud](#) for a philosophical discussion, [my article](#) and [[POSA2](#)] for subtypes of *Monolith*, Martin Fowler's discussion on [starting development with Monolith](#), [[MP](#)] for the [definition of monolithic hell](#) and a post describing the [first-hand experience of it](#).

We distance ourselves from the [systems architecture's definition](#) of *Monolith* as a single unit of deployment because our main focus lies with the internal structure of systems.

Instead, we will use the old definition of a *monolithic* application as a cohesive lump of code containing no discernible components [[GoF](#), [POSA1](#)].

A *Monolith* is non-modular (not divided by interfaces) along all the structural dimensions. Its thorough cohesiveness is both its blessing (single-step debugging, system-wide optimizations) and its curse (messy code, no scalability of development and deployment, zero flexibility).

Performance

On one hand, monolithic applications provide perfect opportunities for performance optimizations as every piece of code is readily accessible from any other. On the other hand, if the application is stateful, access to the state may [limit the performance benefit](#) of using multiple CPU cores. Furthermore, large *Monoliths* may become too messy for programmers to identify and too complicated and fragile to implement any non-local optimizations that could drastically improve performance.

There are many kinds of bottlenecks which limit an application's performance. As soon as you change your code to use multiple CPU cores you may find that the program's throughput [is constrained](#) by the speed of your hard drive or network interface. And when you upgrade those two, you may well hit something more subtle, like OS interrupts or [CPU cache coherence](#).

Overall, tiny *Monoliths* provide the best latency and throughput per CPU core. Larger performance-critical projects may need to partition the code into [Layers](#) or [Services](#) so that any manually optimized part remains small enough to be manageable. Higher throughput is attainable through distributing the software over multiple computers: [Shards](#) employ several copies of the whole system while a [Pipeline](#) may run each step of data processing on a separate server.

Dependencies

Even though a *Monolith* is a single module, meaning that there are no dependencies among its parts (in fact, everything depends on everything), it still may depend on some external components or services which it uses. Those dependencies tend to cause [vendor lock-in](#) or make the software OS- or hardware-dependent. [Hexagonal Architecture](#) (including [MVP](#) and [MVC](#)) decouples a monolithic implementation from its dependencies by isolating the latter behind [Adapters](#).

Applicability

Monolith is good for cases which are harmed by the introduction of modularity:

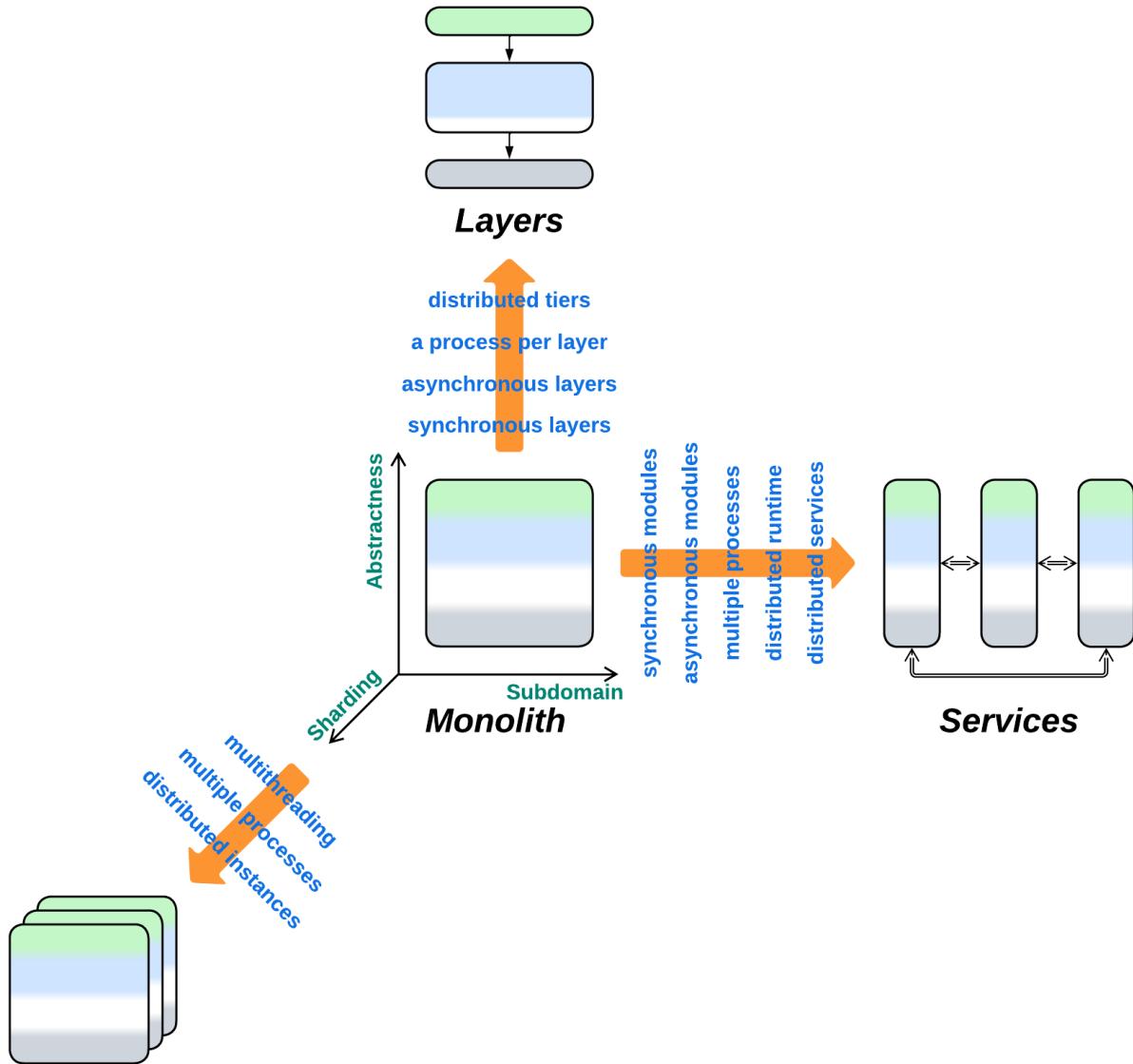
- *Tiny projects*. The project is relatively small (below 10 000 lines) and the requirements will never change (e.g. you need to implement an application for running a specific mathematical calculation or a library supporting a well-established communication protocol).
- *Ultra optimization*. You already have a working and thoroughly optimized system, but you still need that extra 5% performance improvement achievable through merging all the components together.

- *Low latency.* If you need ultra low latency for the entire application, any asynchronous communication between its modules is not a viable option. Example: high-frequency trading.
- *Prototyping.* You are writing a prototype in a domain you are not familiar with, gathering requirements in the process. Chances for a correct initial identification of weakly coupled subdomains (to become modules or services) are [quite low](#) and it is worse to have wrong module boundaries than to use no modules at all. [At the later stages](#) of the project, when you will know the domain much better and your users will have approved the initial implementation, you will be able to split the system into components in a much better way, if and when that will be needed. Nevertheless, you may already know enough to apply [Layers](#) or [Hexagonal Architecture](#) which keep the business logic monolithic while isolating it from the periphery and third-party libraries.
- *Quick and dirty.* You are out of time and money and need to show your customers something right now. There is no time to think, no money to perfect the code, and no day after tomorrow.

Monolith [should be avoided](#) when we need modules:

- *Incompatible forces.* There are [conflicting forces](#) (non-functional requirements) for different subsets of functionality. They require splitting the system into (usually asynchronous) components each of which is specifically designed to satisfy its own subset of forces. Your main tool is the careful selection of appropriate technologies and architectures on a per component basis which may allow the project to satisfy all the non-functional requirements even if the task looks impossible during the initial analysis.
- *Long-running projects.* The project is going to evolve over time and you believe you can predict the general direction of the future changes. Modularity brings flexibility which you will need for sure.
- *Larger codebases.* The project grows above average size (100 000 lines of code). If you don't split it into smaller components it will grow into a [monolithic hell](#) with development and debugging slowing down year after year till it reaches [terminal stage](#). Slow development is a waste of money, both in salary and in time to market.
- *Multiple teams.* You have multiple teams to work on the project. Inter-team communication is hard and error-prone whereas merging several teams together is known to greatly reduce the programmers' productivity (which peaks with teams of 5 or less members). Explicit interfaces between components will formalize interdependencies between the teams, lowering communication overhead.
- *Fault tolerance.* Your domain requires fault tolerance which is next to impossible for large monolithic applications.
- *Resource-limited.* Your project is too resource-hungry for commodity hardware. Even if you buy the best server for its needs right now, it is going to crave more tomorrow (or on the next Black Friday).
- *Distributed setup.* Your project needs to run on multiple hardware devices. One of common examples is a [web service](#) containing frontend and backend.

Relations



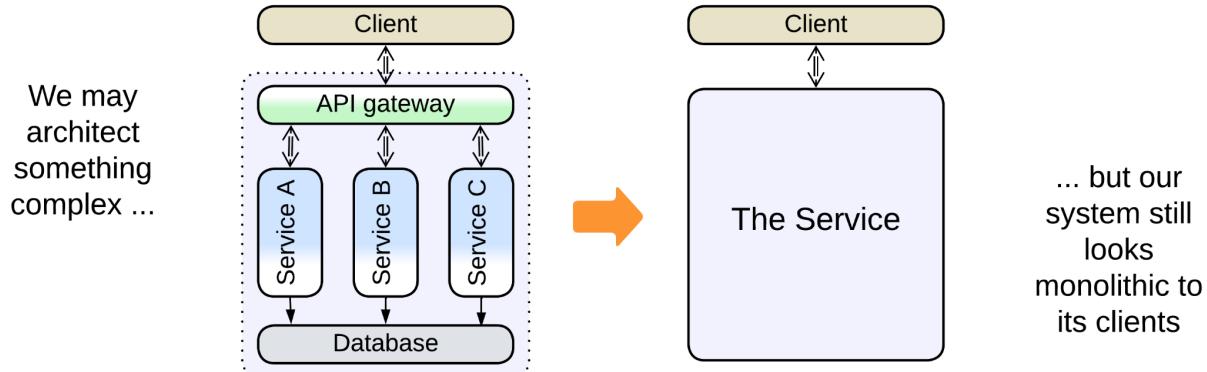
Shards

Monolith:

- Can be extended with a [Proxy](#), or turned into a [Hexagonal Architecture](#) or [Plugins](#).
- Yields [Layers](#), [Services](#), or [Shards](#) if divided along the *abstraction*, *subdomain*, or *sharding* dimensions, correspondingly. All the known architectures are combinations of those three metapatterns.
- Is the bird's-eye view of any architecture.

Variants by the internal structure

Monoliths are the atoms to create more complex architectures from, the opaque building blocks, each of which satisfies a consistent set of forces. Any individual component of a more complex architecture either is monolithic or encapsulates another architectural pattern, decomposable into *Monoliths*, and any architecture looks monolithic to its clients.



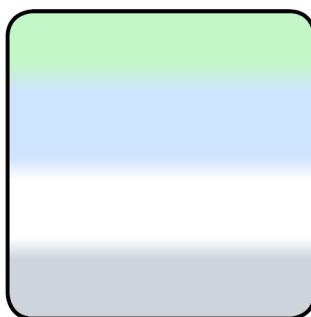
There is a misunderstanding because *software architecture* inspects the internals of *applications* at the level of *modules* or even *classes* while *systems architecture* deals with *distributed systems* and operates *deployment units* which tend to incorporate multiple modules or even applications. Each branch of the architecture [calls](#) its atomic unit a *Monolith*, leading to the term sticking both to a *module that cannot be subdivided*, as in [\[GoF\]](#) and [\[POSA1\]](#), and to a *(sub)system which must be deployed together*, as in present-day literature.

As we aspire to build a unified classification for both distributed and local systems, we must treat both kinds of components in the same way, whether they are [distributed services](#), [co-located Actors](#), or [in-process modules](#). Thus, for the scope of the current book, we will follow the definition of *Monolith* from [\[GoF\]](#): “Tight coupling leads to *monolithic* systems, where you can't change or remove a class without understanding and changing many other classes”. Still, we need to account for a couple of misnomers from system architecture.

True Monolith, Big Ball of Mud

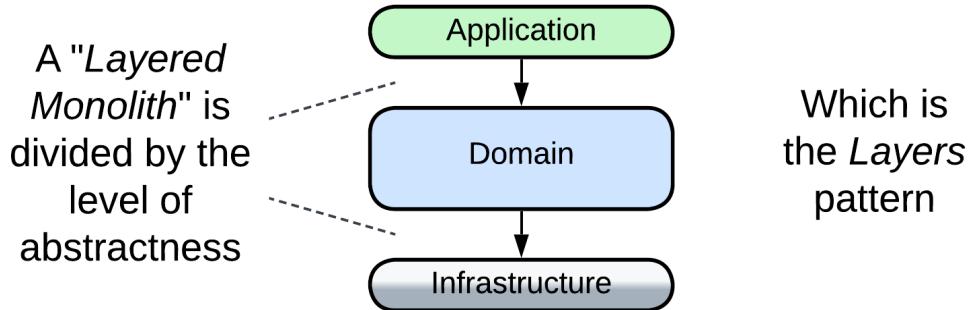
A true
Monolith is
non-modular

"Monolith"
means
"single stone"



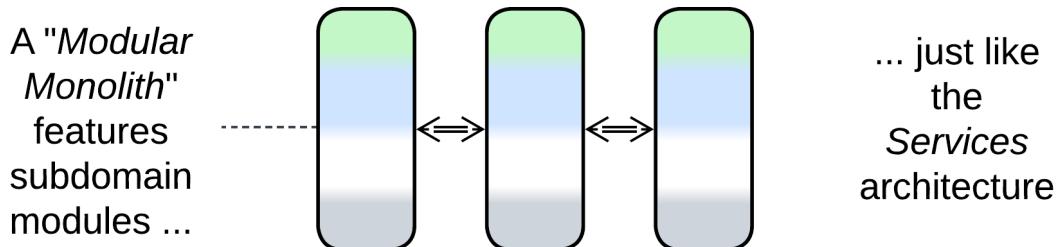
A true *Monolith* features [no clear internal structure](#). If it has any components, they are so tightly coupled that the entire thing behaves as a single cohesive module. This is what we explore in the current chapter.

(misapplied) Layered Monolith



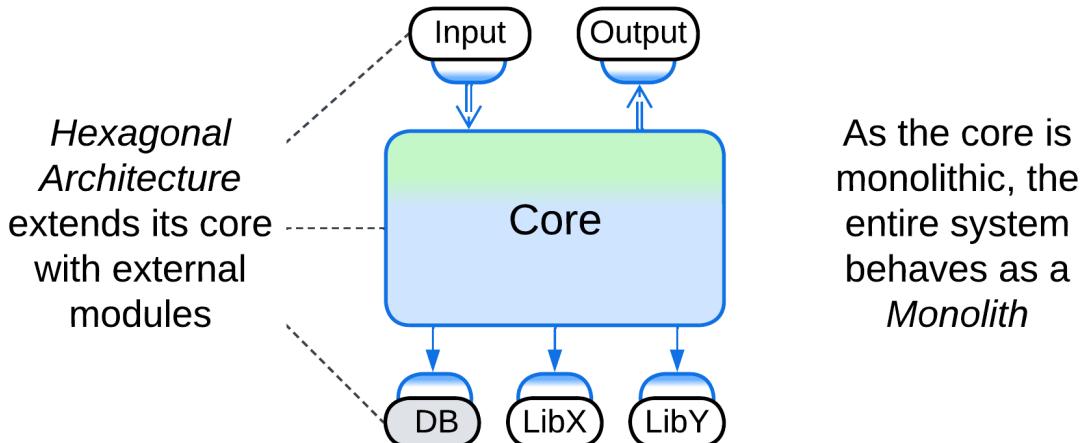
When they say [Layered Monolith \[FSA\]](#), that refers to a non-distributed application with a layered structure, which is a proper [Layers](#) architecture and will be discussed in the corresponding chapter. It is called a *Monolith* for the [sole reason that it is not distributed](#). Nevertheless, *Layers* resemble *Monolith* in many aspects, including easy debugging and the risk of outgrowing the comfort zone of developers.

(misapplied) Modular Monolith (Modulith)



A [Modular Monolith \(Modulith\) \[FSA\]](#) is a single-process application subdivided into modules that correspond to subdomains. If the modules communicate via in-process messaging, the architecture is nearly identical to coarse-grained [Actors](#), thus it is a *Monolith* only in name. *Modulith* [is a kind of Services](#) – it supports development by multiple teams and the asynchronous variant is hard to debug. The relation to *Monolith* is mostly limited to the inability to scale individual parts of the system.

(inexact) Plugins and Hexagonal Architecture



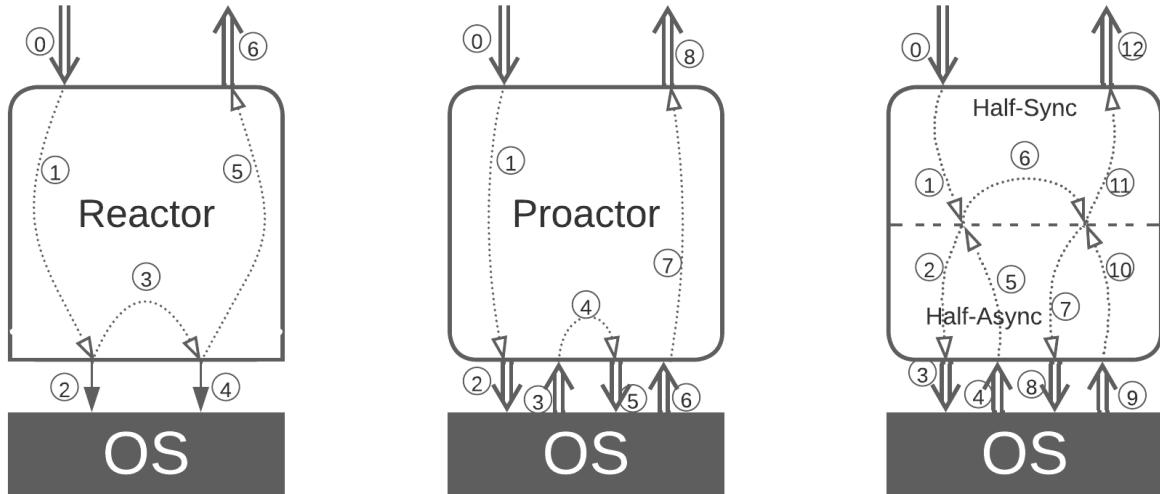
[Plugins \[FSA\]](#) and [Hexagonal Architecture](#) extend a (sub)system with external components. These architectures can be applied to a *Monolith* without drastically changing its properties – it still remains relatively easy to write and debug but hard to support when

outgrown. Therefore, we will not currently discuss these modifications, mainly because each of them has a dedicated chapter.

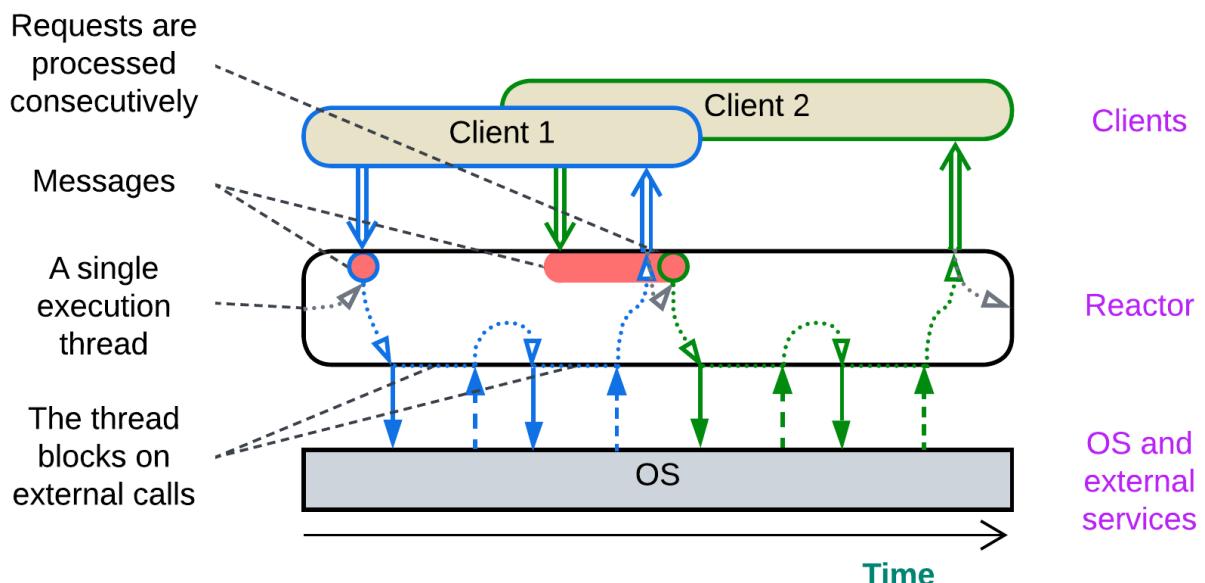
Variants by the mode of action

Let's take a look inside a *Monolith*.

Any software module reacts to incoming events or data and produces outgoing events or data. But there are a few basic ways to implement that cycle:



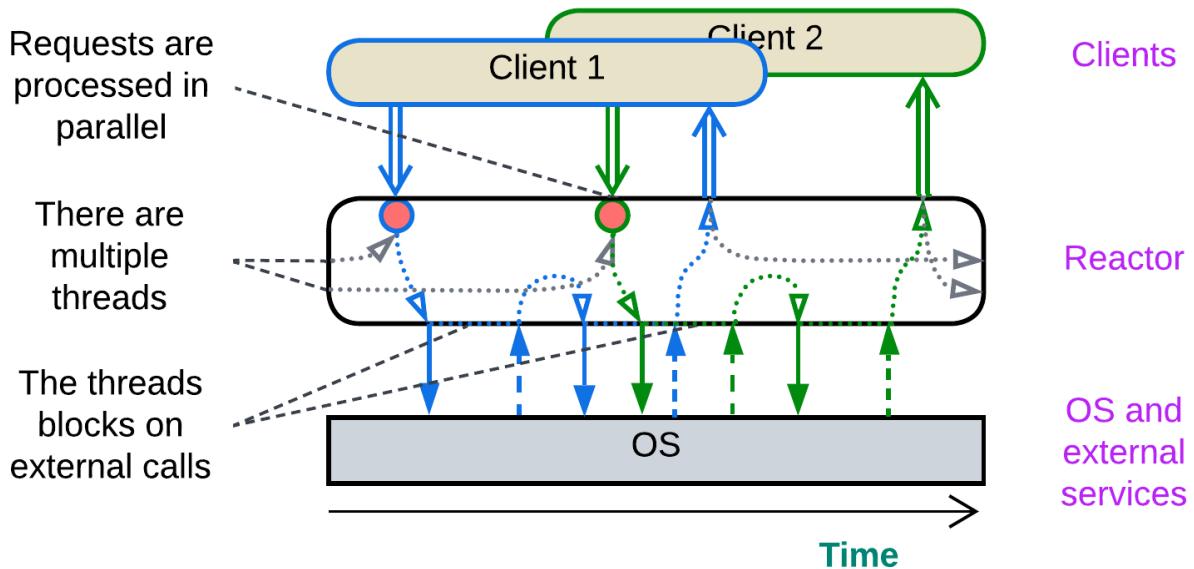
Single-threaded Reactor (one thread, one task)



In a [Reactor](#) [POSA2] a single thread waits for an incoming event or data packet, processes it with blocking calls to the underlying OS, hardware, and external dependencies and returns the result, rinse and repeat.

That makes sense when the module owns and provides access to a hardware component which cannot do several actions at once, for example, a communication bus or a HDD firmware capable of a single read or write at any given moment.

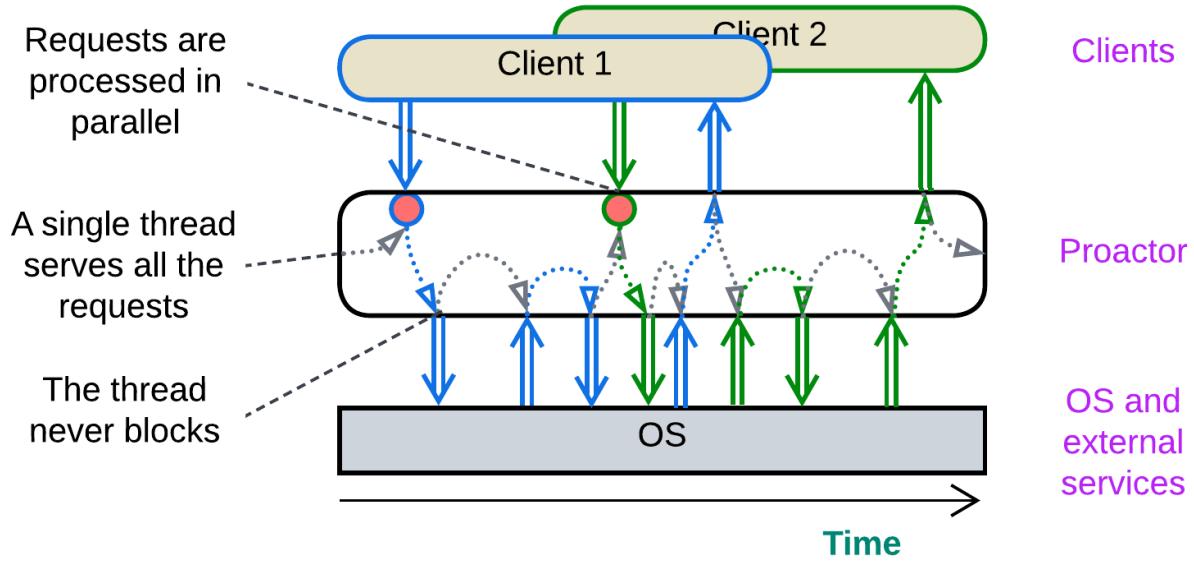
Multi-threaded Reactor (a thread per task)



A [Reactor](#) [POSA2] may employ multiple threads by having a *pool* of them waiting for a request or data to come. The incoming event activates a thread, which becomes dedicated to processing it, does several blocking calls and, finally, sends back a response. When the request processing is complete, the thread returns to the pool of idle threads to wait for the next event to process.

This is the default simple & stupid implementation of backend services. Its pitfalls include contention for shared resources, deadlocks, and high memory consumption by OS-level threads.

Proactor (one thread, many tasks)

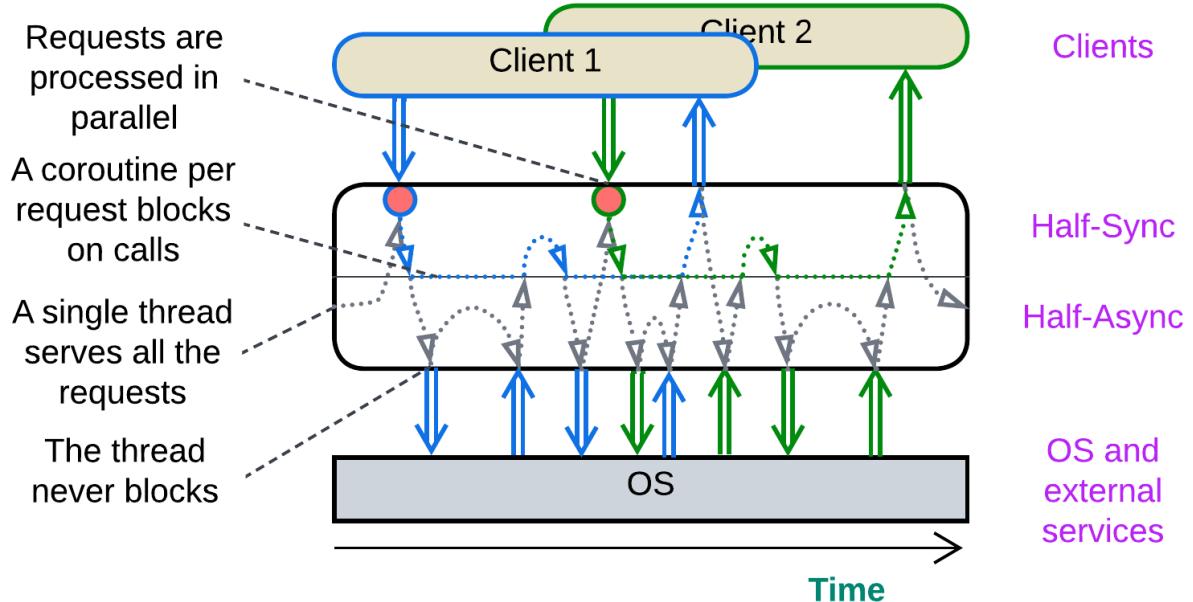


In [Proactor](#) [POSA2] a single thread processes all of the incoming events, both from the module's clients and from the hardware or dependencies it manages. When an event is received, the thread goes through a short piece of corresponding business logic (*event handler*) which usually does one or more non-blocking actions, such as sending messages to other components, writing to registers of the managed hardware, or initiating an async I/O.

As soon as the event handler returns, the thread becomes ready to process further events. As the thread never blocks, it is resource-efficient and serves many interleaved tasks.

This approach is good for real-time systems where thread synchronization is largely forbidden because of the associated delays and for reactive [control](#) applications which mostly adapt to the environment instead of running pre-programmed scenarios. The drawback is very poor structure of the code and debuggability as any complex behavior is broken into many independent event handlers.

(inexact) Half-Sync/Half-Async (coroutines or fibers)



[Half-Sync/Half-Async \[POSA2\]](#) originally described the interaction between user space and kernel threads in operating systems which is not much different from that behind coroutines and fibers. A single thread (or a thread pool with one thread per CPU core) handles all the incoming events and switches its call stack in the process.

Every incoming request is allocated a call stack which stores the processing state (local variables and methods called) of the request. When it needs to access an external component, the [runtime system](#) saves the request's stack, does a non-blocking call, and the execution thread returns to its original stack to wait for any new event to handle while the request processing stack remains frozen until the action it has initiated completes asynchronously. Then the runtime switches the execution thread back to the stored request's stack and continues processing the request until it completes and its stack is deleted.

This makes programming and debugging feel as easy as they are with [Reactor](#) (imperative style) while retaining the low resource consumption and high performance of [Proactor](#) (reactive paradigm). Coroutines and fibers are used in highly efficient [game engines](#) and [databases](#). Though *Half-Sync/Half-Async* has two layers (is not truly monolithic), I believe it belongs next to *Reactor* and *Proactor* which make up its upper and lower halves, correspondingly.

The state of the art

These patterns are not widely known and programmers tend to mix them together, for better or for worse. One is likely to encounter a heavily multithreaded big ball of mud where some threads serve user requests while others are dedicated to periodic service routines.

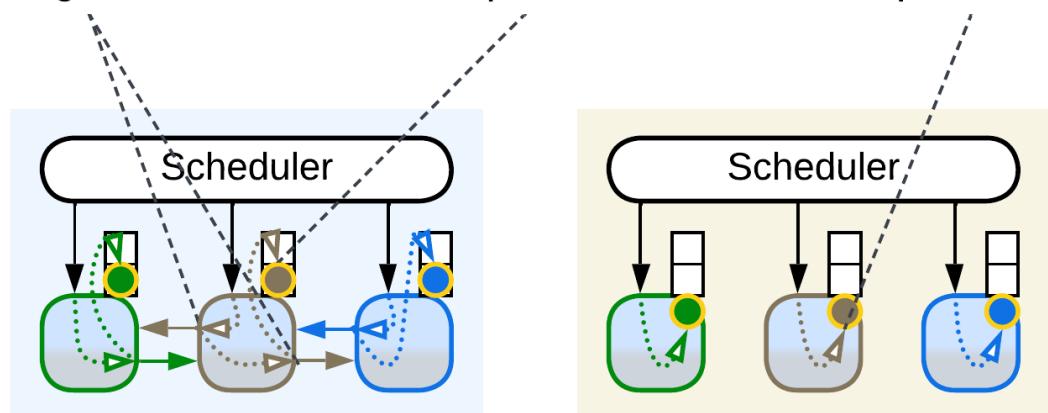
Moreover, people [often call](#) any event-driven service *Reactor*, causing confusion among those who distinguish between the three patterns.

(inexact) (Re)Actor-with-Extractors (phased processing)

During an extract phase each object queries its neighbors

It plans actions and puts them on its message queue

In a react phase objects execute the actions they planned



Extract Phase

React Phase



Extract and react phases alternate

As a bonus, let's review an [unconventional execution model](#) that fits game development or other kinds of simulation with many interacting objects.

We have a long-running system where each simulated object with a complex behavior depends on the objects around it. Common wisdom proposes two ways to implement it:

- [Actors](#) (asynchronous messaging, reactive programming) – each *actor* (simulated object) runs single-threaded and wakes up only to process incoming messages. While processing a message, an actor may change its state and/or send messages to other actors. The entire actor's data is private and there are no synchronous calls between the actors. The good thing is that actors are very efficient in highly parallel tasks as there are no locks in their code. The bad thing is that actors have no way to synchronize their states: you can only request another actor to tell you about its state, and its response may become outdated even before you receive it. Also, any complex logic that involves multiple actors is fragmented into many event handlers.
- The opposite approach is to have the simulated objects access each other synchronously. This allows for complex logic that depends on states of several

objects but gets in trouble with changing the objects' states from multiple threads: you need to protect them with those inefficient locks and you get those dreadful deadlocks as the outcome.

Here we see two bad options to choose from. However, it is the simulated nature of the system that saves the day: we can *stop the world to get off*. The objects' querying each other and their changing their states neither needs to happen at the same time nor obey the same rules!

The simulation runs in steps. Each step consists of two phases:

- *Query phase (extraction)* is when the object states are immutable, thus the objects can communicate synchronously with no need for locks. In this phase each object collects information from its surroundings (other objects), plans its actions and posts them as commands to its own message queue. I suppose that objects may also send commands to each other in this phase.
- *Command phase (reaction)* is when each object executes its planned (queued) actions that change its state, but it cannot access other objects.

Each phase lasts until every object in the system completes its tasks scheduled for that particular phase. The phase toggle is supervised by a [Scheduler](#) which runs the objects on all the available CPU cores. The entire process resembles the [game of Mafia](#) with public daily conversations and covert nightly actions.

(Re)Actor-with-Extractors is the perfect example of earning the benefits of two architectures without paying the penalties. It utilizes both the lockless parallelism of Actors-style [shared-nothing](#) and the simplicity of synchronous access in [shared-memory](#) by alternating between those two modes through applying the [CQRS principle](#) to the time dimension.

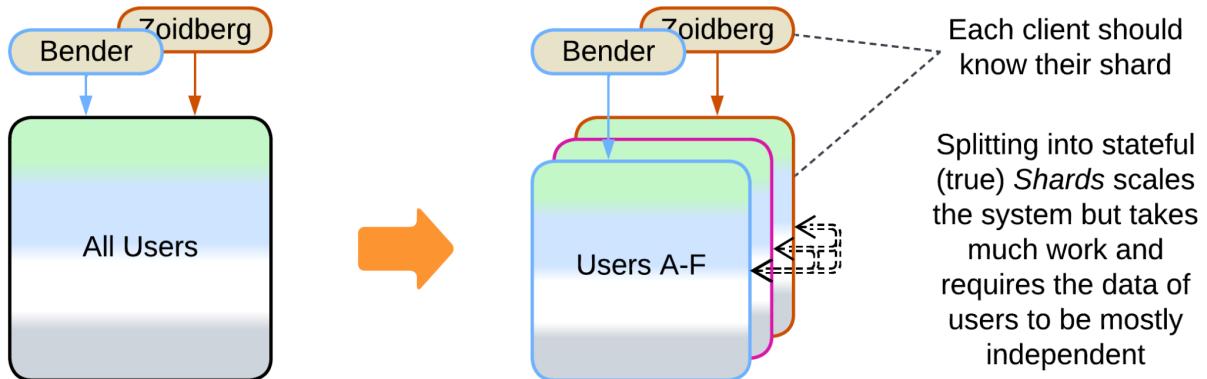
Evolutions

Every architecture has drawbacks and tends to evolve in a variety of ways to address them. Below is a brief summary of common evolutions of *Monolith* with more information available in [Appendix E](#).

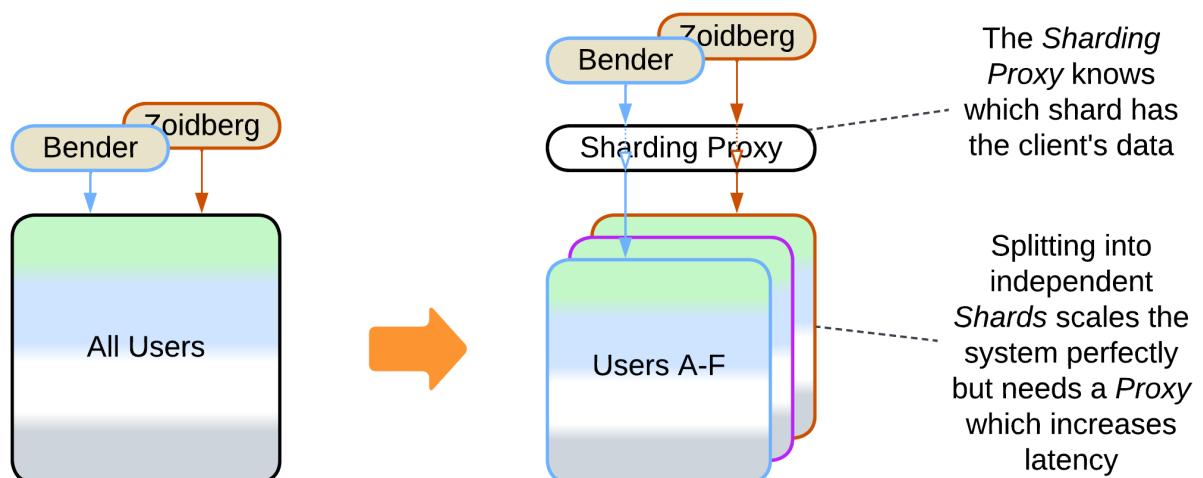
Evolutions to Shards

One of the main drawbacks of monolithic architecture is its lack of scalability – a single running instance of your system may not be enough to serve all the clients no matter how many resources you add in. If that is the case, you should consider [Shards](#) – *multiple instances* of a *Monolith*. There are following options:

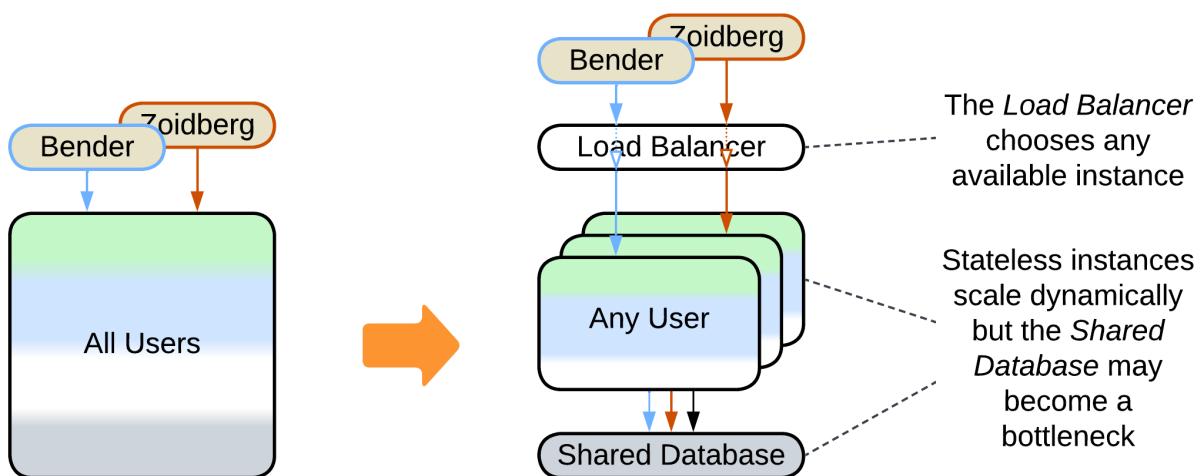
- Self-managed [Shards](#) – each instance owns a part of the system's data and may communicate with all the other instances (forming a [Mesh](#)).



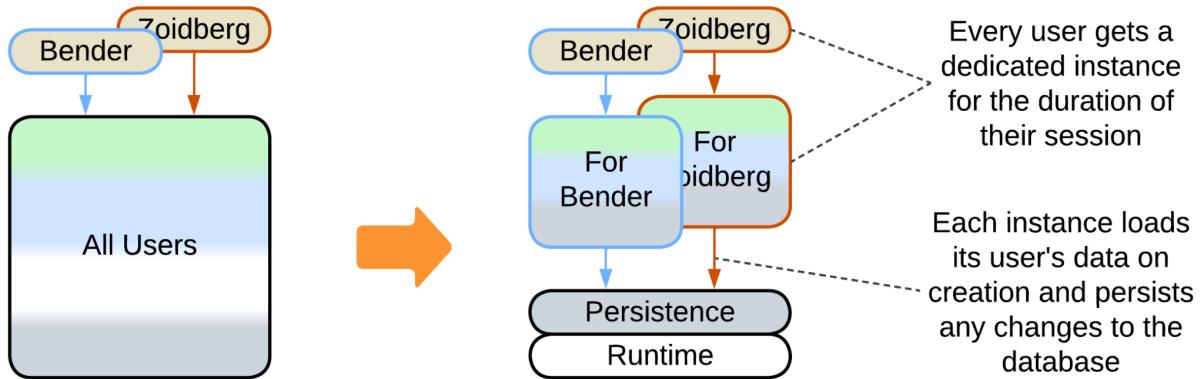
- *Shards with a Sharding Proxy* – each instance owns a part of the system's data and relies on an external component to choose a shard for a client.



- A *Pool* of stateless instances with a *Load Balancer* and a *Shared Database* – any instance can process any request, but the database limits the throughput.



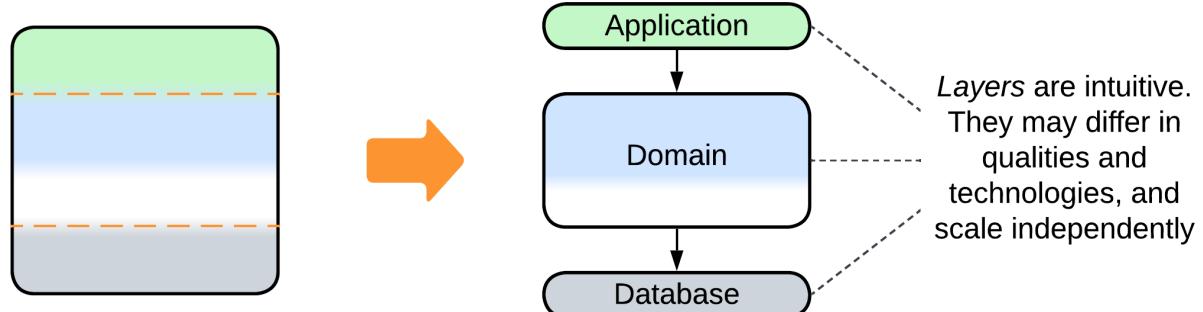
- A *Stateful Instance* per client with an external persistent storage – each instance owns the data related to its client and runs in a virtual environment (i.e. web browser or an *Actor Framework*).



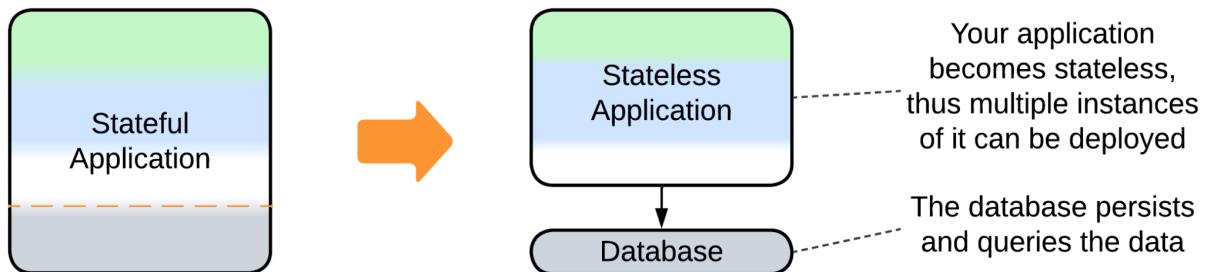
Evolutions to Layers

Another drawback of *Monolith* is its... er... monolithism. The entire application exposes a single set of qualities and all its parts (if they ever emerge) are deployed together. However, life awards flexibility: parts of a system may benefit from being written in varying languages and styles, deployed with different frequency and amount of testing, sometimes to specific hardware or end users' devices. They may need to [vary in security and scalability](#) as well. Enter [*Layers*](#) – a subdivision by the *level of abstractness*:

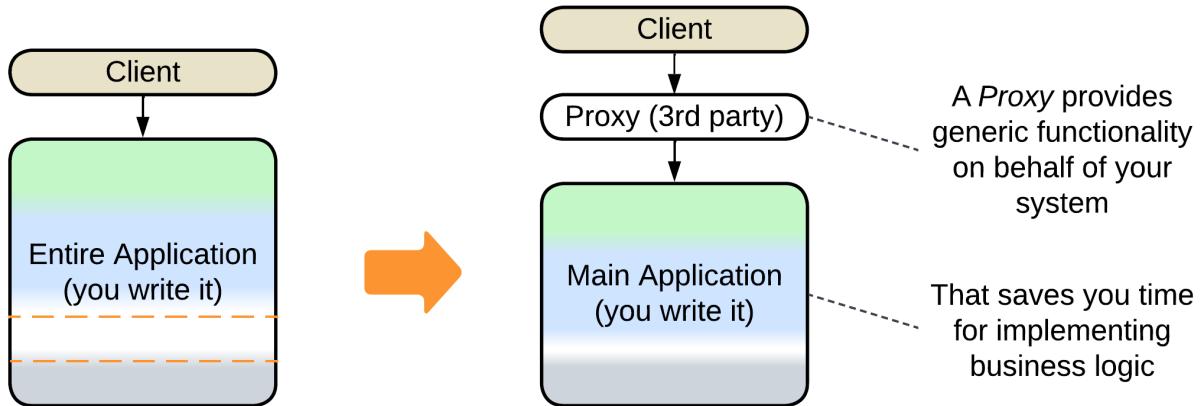
- Most *Monoliths* can be divided into 3 or 4 layers of different abstractness.



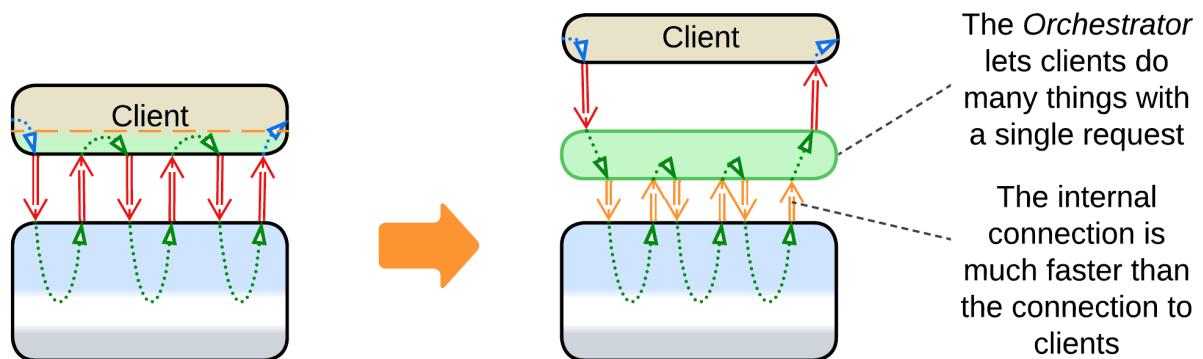
- It is common to see the database separated from the main application.



- [*Proxies*](#) (e.g. [Firewall](#), [Cache](#), [Reverse Proxy](#)) are common additions to the system.



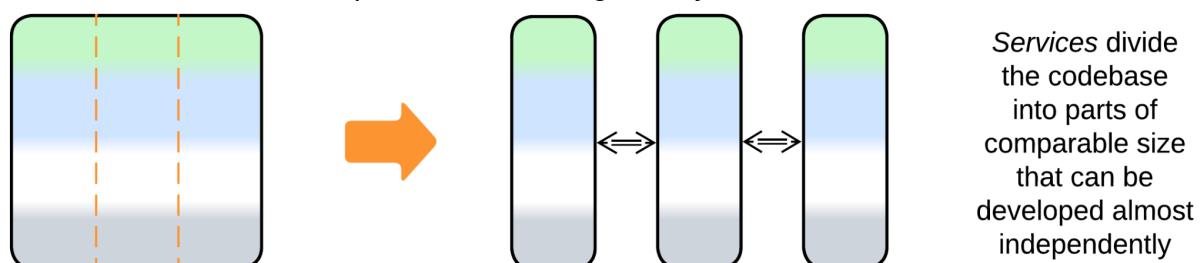
- An [Orchestrator](#) adds a layer of indirection to simplify the system's external API.



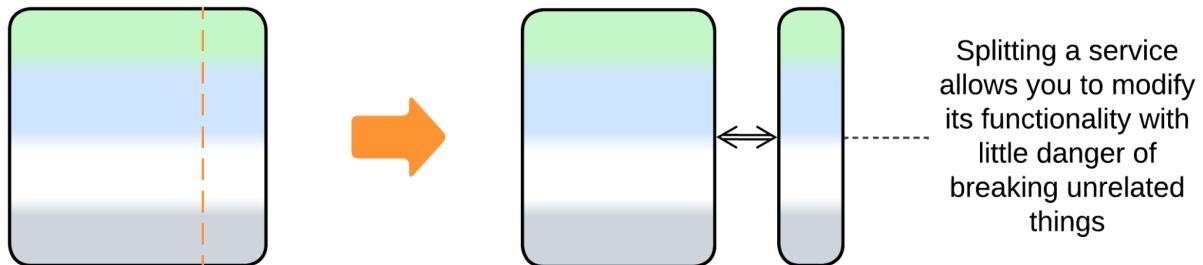
Evolutions to Services

The final major drawback of *Monolith* is the cohesiveness of its code. The rapid start of development with *Monolith* begets a major obstacle as the project grows: every developer needs to know the entire codebase to be productive while changes made by individual developers overlap and may break each other. Such distress is usually solved by dividing the project into modules along *subdomain boundaries* (which usually match [bounded contexts](#)). However, that requires much work, and good boundaries and APIs are hard to design. Thus many organizations prefer a slower iterative transition.

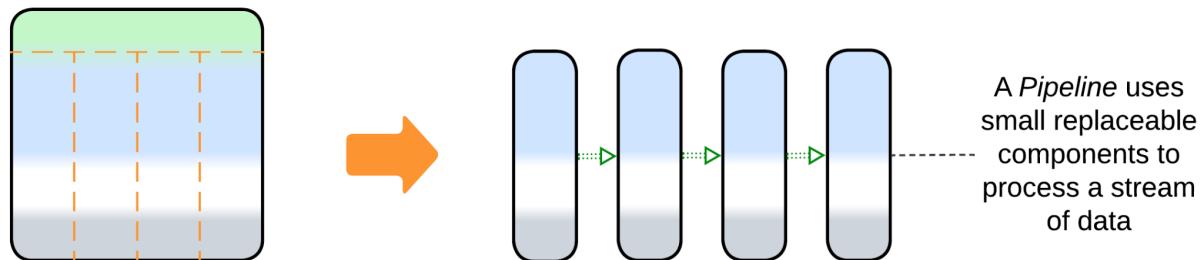
- A *Monolith* can be split into [Services](#) right away.



- A feature may be added or a weakly coupled part separated into a new service.



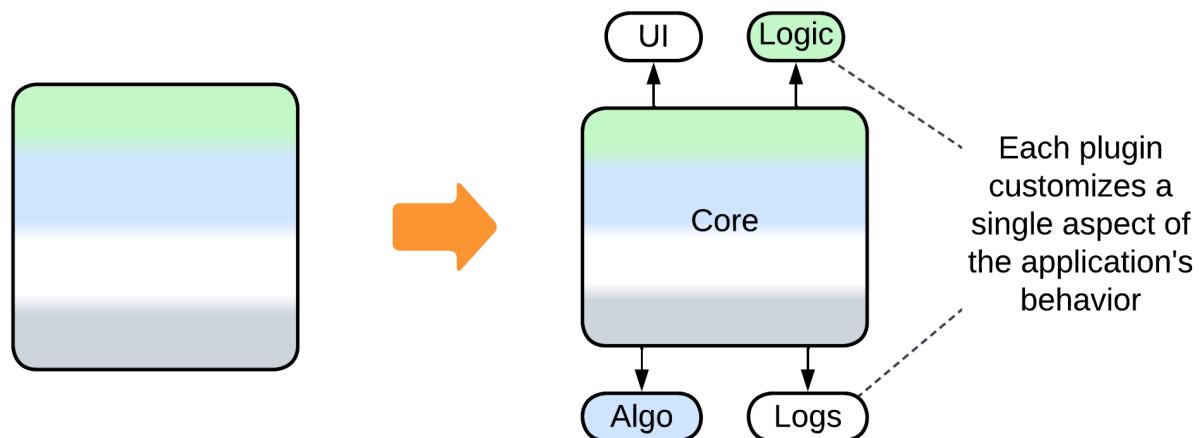
- Some domains allow for sequential data processing best described by [Pipelines](#).



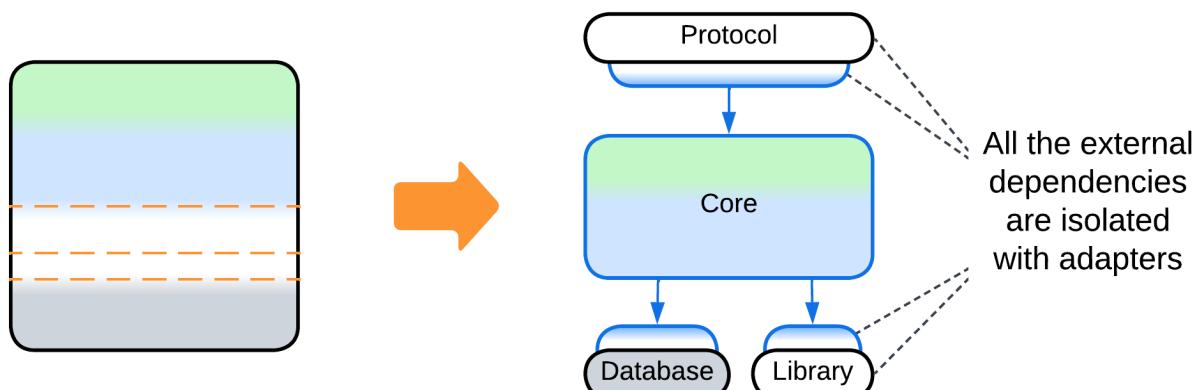
Evolutions with Plugins

The last group of evolutions does not really change the monolithic nature of the application. Instead, its goal is to improve the customizability of the *Monolith*:

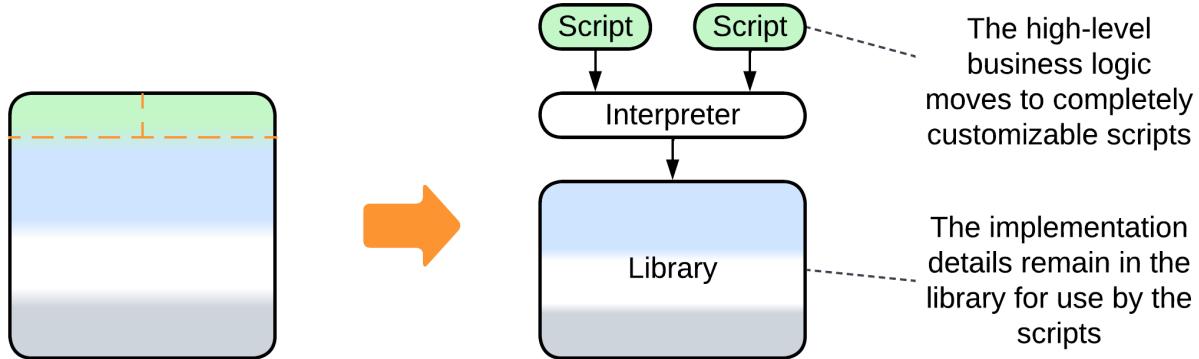
- Vanilla [Plugins](#) is the most direct approach which relies on replaceable bits of logic.



- [Hexagonal Architecture](#) is a subtype of *Plugins* which is all about isolating the main code from any third-party components it uses.



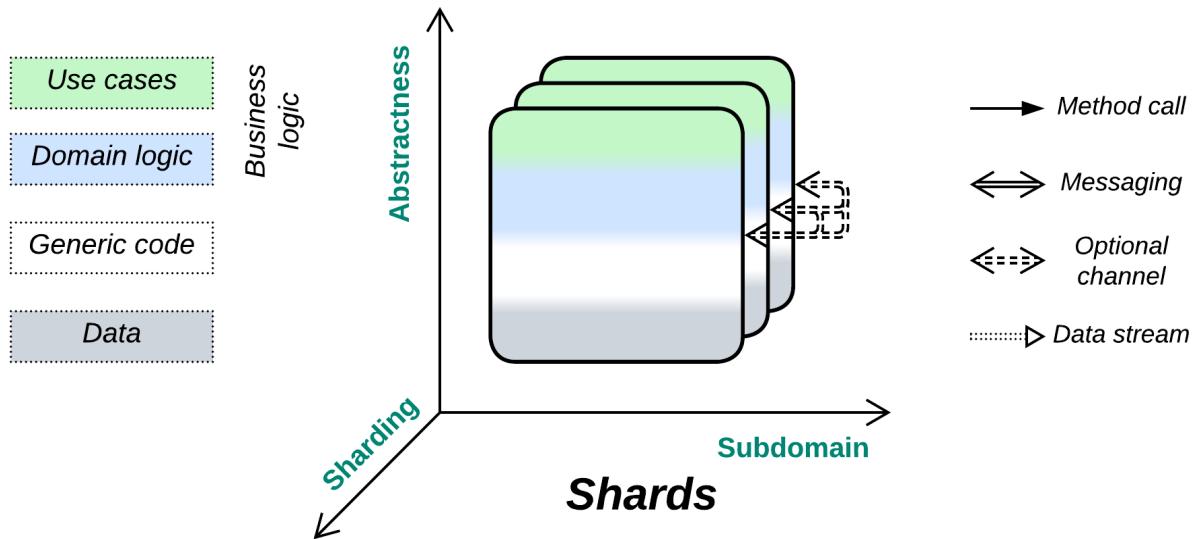
- *Scripts* is a kind of *Microkernel* – yet another subtype of *Plugins* – which gives users of the system full control over its behavior.



Summary

A *Monolith* is an unstructured application. It is the best architecture for rapid prototyping by a small team and it usually grants the best performance to costs ratio. However, it does not scale, lacks any flexibility and becomes unmanageable as the amount of code grows.

Shards



Attack of the clones. Solve scalability in the most straightforward manner.

Known as: Shards, Instances, Replicas [[DDS](#)].

Variants:

By isolation:

- Multithreading,
- Multiple processes,
- Distributed instances.

By state:

- Persistent slice: [Sharding](#) / Shards [[DDS](#)] / Partitions [[DDIA](#)] / Cells ([Amazon definition](#)),
- Persistent copy: Replica [[DDS](#)],
- Stateless: Pool [[POSA3](#)] / Instances / Replicated Stateless Services [[DDS](#)] / Work Queue [[DDS](#)],
- Temporary state: Create on Demand.

Structure: A set of functionally identical subsystems with little or no intercommunication.

Type: Implementation.

Benefits	Drawbacks
Good scalability	It's hard to synchronize the system's state
Good performance	

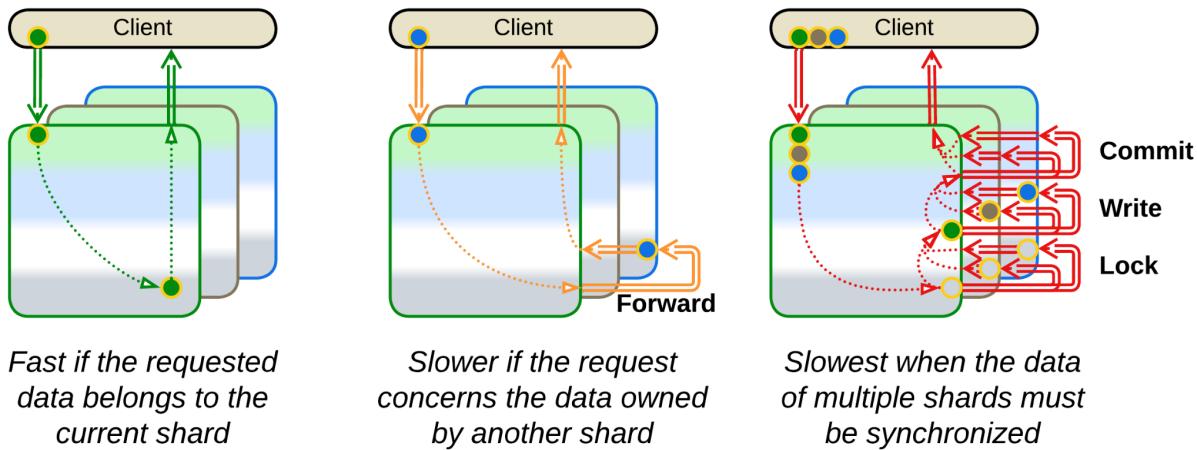
References: [[POSA3](#)] is dedicated to pooling and resource management; [[DDS](#)] reviews *Shards, Replicas and Stateless Instances*; [[DDIA](#)] covers sharding and synchronization of *Replicas* in depth; Amazon promotes full-system sharding as [Cell-Based Architecture](#).

Shards are multiple and, in most cases, independent instances of a component or subsystem which the pattern is applied to. They provide scalability, often redundancy and sometimes locality, at the cost of slicing or duplicating the component's state (writable data), which obviously does not affect inherently stateless components. Most of this pattern's specific evolutions look for a way to coordinate shards at the logic or data level.

There is a sibling metapattern, [Mesh](#), in which instances of a component closely communicate among themselves. The difference between the patterns lies in the strength of interactions: while each *shard* exists primarily to serve its clients, a *Mesh node*'s priority is preserving the *Mesh* itself from falling prey to entropy, making the *Mesh* into a reliable distributed (virtual) layer. Some systems, such as distributed databases, hold the middle ground – their shards or nodes both intercommunicate intensely and execute a variety of client requests.

Performance

A *shard* retains the performance of the original subsystem (a [Monolith](#) in the simplest case) as long as it runs independently. Any task that involves intershard communication has its performance degraded by data serialization and network latency. And as soon as multiple shards need to synchronize their states you find yourself on the horns of a dilemma: damage data consistency through write conflicts or kill performance with distributed transactions [[FSA](#)].



A [Shared Repository](#) (or its derivation, the [Space-Based Architecture](#)) is a common solution to let multiple shards access the same dataset. However, it does not solve the performance vs consistency conflict (which is rooted in the [CAP theorem](#)) but only encapsulates its complexity inside a ready-made third-party component, making your life easier.

Dependencies

You may need to make sure that all the shards are instances of the same version of your software or at least that their interfaces and contracts are backward- and [forward-compatible](#).

Applicability

A *sharded* system features properties of a pattern it replicates (a single-component [Monolith](#), local [layered](#) application or distributed [Services](#)). Its peculiarities that originate with the *Shards' scalability*, are listed below.

Shards are good for:

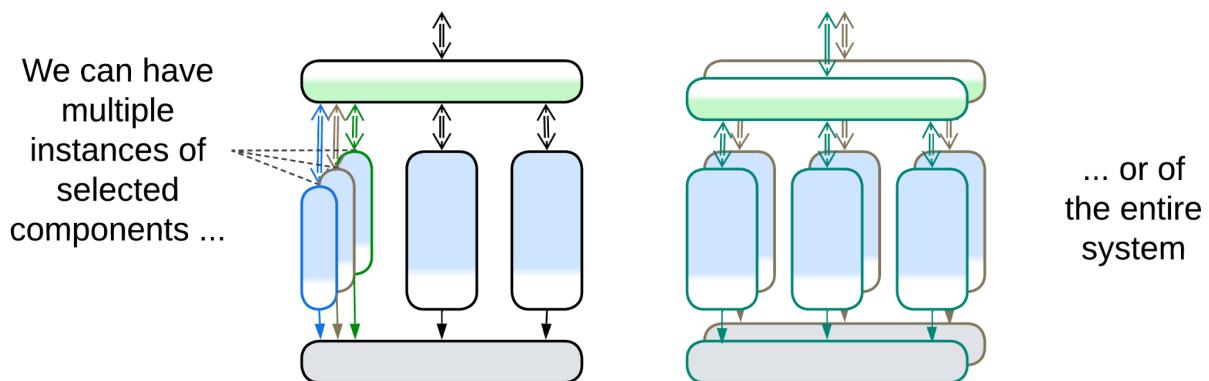
- *High or variable load.* You need to scale your service up (and sometimes down). With *Shards* you are not limited to a single server's CPU and memory.
- *Survival of hardware failures.* A bad HDD or failing RAM does not affect your business if there is another running instance of your application. Still, make sure that your [Load Balancer](#) and Internet connection are replicated as well.

- *Improving worst case latency*. If your service suffers from latency spikes, you can run a few replicas of it in parallel, broadcasting every user request to all of them, and returning the fastest response. Adding a single replica turns your p90 into [p99](#).
- *Improving locality*. A world-wide business optimizes latency and costs by deploying an instance of its software to a local data center in every region of interest (or even to its clients' browsers!).
- [*Canary Release*](#). It is possible to deploy an instance of your application featuring new code along with the old, stable instances. That tests the update in production.

Shards' weak point is:

- *Shared data*. If multiple instances of your application need to modify the same dataset, that means that none of them properly owns the data, thus you have to rely on an external component (a *data layer*, implying [Layers](#) and [Space-Based Architecture](#) or another kind of [Shared Repository](#)).

Relations



Shards:

- Applies to a [Monolith](#) or any kind of subsystem.
- Can be extended with [Middleware](#), [Shared Repository](#), [Proxies](#) or [Orchestrator](#).
- Is the foundation for [Mesh](#).

Variants by isolation

There are intermediate steps between a single-threaded component and distributed *Shards* which gradually augment the pros and cons of having multiple instances of a subsystem:

Multithreading

The first and very common advance towards scaling a component is running multiple execution threads. That attempts to utilize all the available CPU cores or memory bandwidth but [requires](#) protecting the data from simultaneous access by several threads, which in turn may cause deadlocks.

Benefits	Drawbacks
Limited scalability	More complex data access

Multiple processes

The next stage is running several (usually single-threaded) instances of the component on the same system. If one of them crashes, others survive. However, sharing data among them and debugging multi-instance scenarios becomes non-trivial.

Benefits	Drawbacks
Limited scalability	Non-trivial shared data access
Software fault isolation	Troublesome multi-instance debugging

Distributed instances

Finally, instances of the subsystem may be distributed over a network to achieve nearly unlimited scalability and fault tolerance by [sacrificing](#) the consistency of the whole system's state.

Benefits	Drawbacks
Full scalability	No shared data access
Full fault isolation	Hard multi-instance debugging
	No good way to synchronize state of the instances

Variants by state

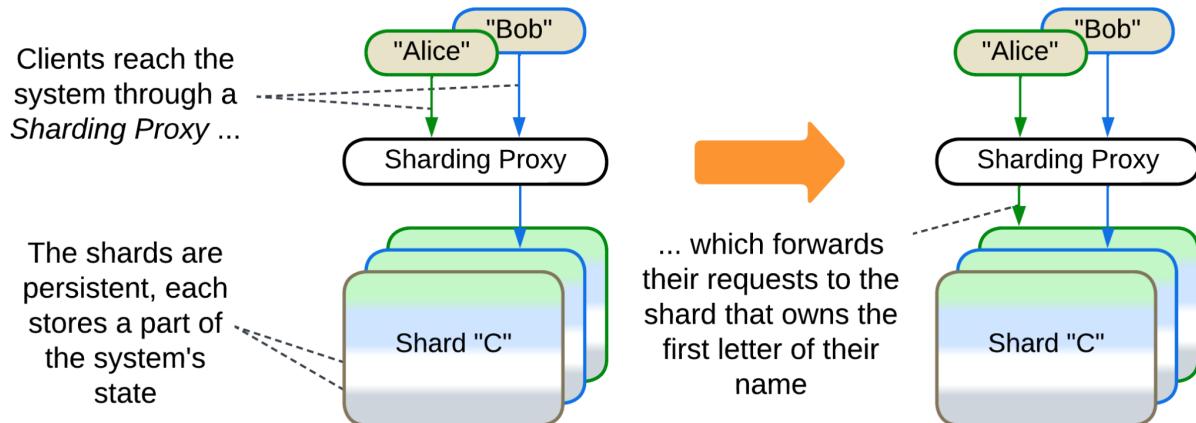
Sharding can often be transparently applied to individual components of [data processing](#) systems. That does not hold for [control systems](#) which need centralized decisions based on the modeled system's state, which must be accessible as a whole, thus the main business logic that owns the model (last known state of the system) cannot be sharded.

Many kinds of *Shards* require an external coordinating module ([Load Balancer](#)) to assign tasks to the individual instances. In some cases the coordinator may be implicit, e.g. an OS socket or scheduler. In others it may be replicated and co-located with each client (as an [Ambassador \[DDS\]](#)).

Shards usually don't communicate with each other directly. The common exception is [Mesh](#) which includes distributed databases and [actor](#) systems that explicitly rely on communication between the instances.

There are several subtypes of sharding that differ in the way they handle state:

Persistent slice: Sharding, Shards, Partitions, Cells (Amazon definition)



Shards [DDS] own the non-overlapping parts of the system's state. For example, a sharded phonebook (or DNS) would use one shard for all contacts with initial "A", another shard for contacts with initial "B" and so on (in reality they use hashes [DDIA]). A large wiki or forum may run several servers, each storing a subset of the articles. This is proper *sharding*, which is also called *partitioning* [DDIA] in the world of databases.

Names are not evenly distributed among letters. Many names start with A but few start with Q. If we use the first letter of a user's name to assign them to a shard, the shard that serves users whose names start with A will be much more loaded than the one responsible for the letter Q. Therefore, real-world systems rely on *hashing* [DDIA] – calculation of a *checksum* of the user's name which yields a seemingly random number. Then we divide the checksum by the total number of shards we have and use the remainder as the id of the shard that has the user's data. For example, $\text{CRC16}(\text{"Bender"}) = 52722$. If we have 10 shards, Bender goes to $(52722 \% 10 = 2)$ the 3rd one.

Cells, according to the [Amazon terminology](#), are copies of a whole system deployed to several data centers, each serving local users. The locality improves latency and saves on Internet traffic while having multiple instances of the system up and running provides availability. The downside of this approach is its complexity and amount of global traffic needed to keep the Cells in sync.

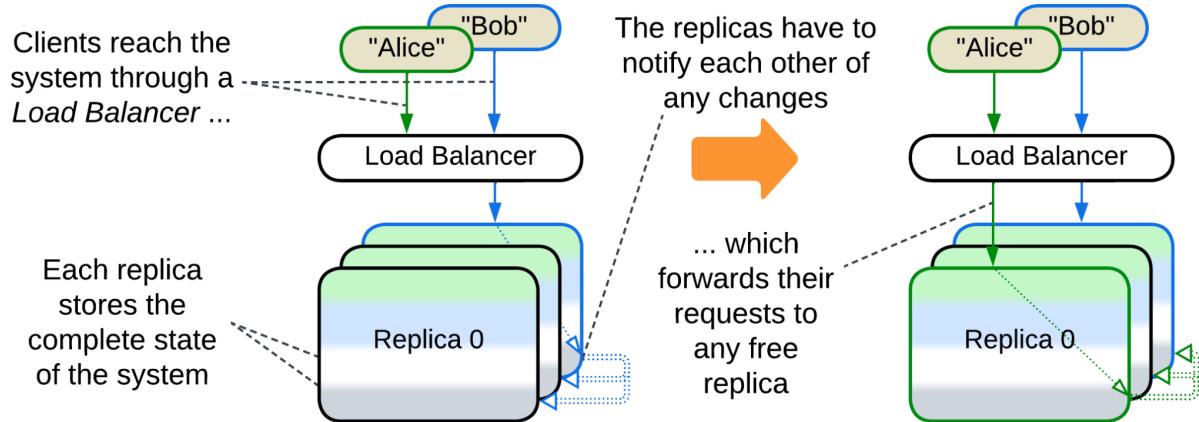
It usually takes a stand-alone [*Sharding Proxy* \[DDS\]](#) – a kind of *Load Balancer* – to route client's requests to the shard that owns its data. However, there are other options [DDIA]:

- The *Sharding Proxy* may be deployed as a client-side [*Ambassador* \[DDS\]](#) to avoid the extra network hop. This approach requires a means for keeping the Ambassadors up-to-date with your system's code.
- You can publish your *sharding function* [DDS] and the number of shards in your public API to let your clients choose which shard to access without your help. That may work for internal clients implemented by your or neighbor team.
- Finally, each shard may be able to forward client requests to any other shard – making each shard into a *Sharding Proxy* and an entry point into the resulting [*Mesh*](#). If your client accesses a wrong shard, the request is still served, though a little slower, through being forwarded between the shards [DDIA].

Sharding solves scaling of an application both in regard to the number of its clients and to the size of its data. However, it works well only if each client's data is independent from

other clients. Moreover, if one of the shards crashes, the information it owns becomes unavailable unless *replication* (see below) has been set up as well.

Persistent copy: Replica



Replicas [DDS] are identical copies of a stateful (sub)system. Replication improves the system's throughput (as each replica serves client requests) and its stability (as a fault in one replica does not affect others which may quickly take up the failed replica's clients). Replicas may also be used to improve tail latency through *Request Hedging*: each request is sent to several replicas in parallel and you use the first response which you receive. Mission-critical hardware [runs three copies](#) and relies on majority voting for computation results.

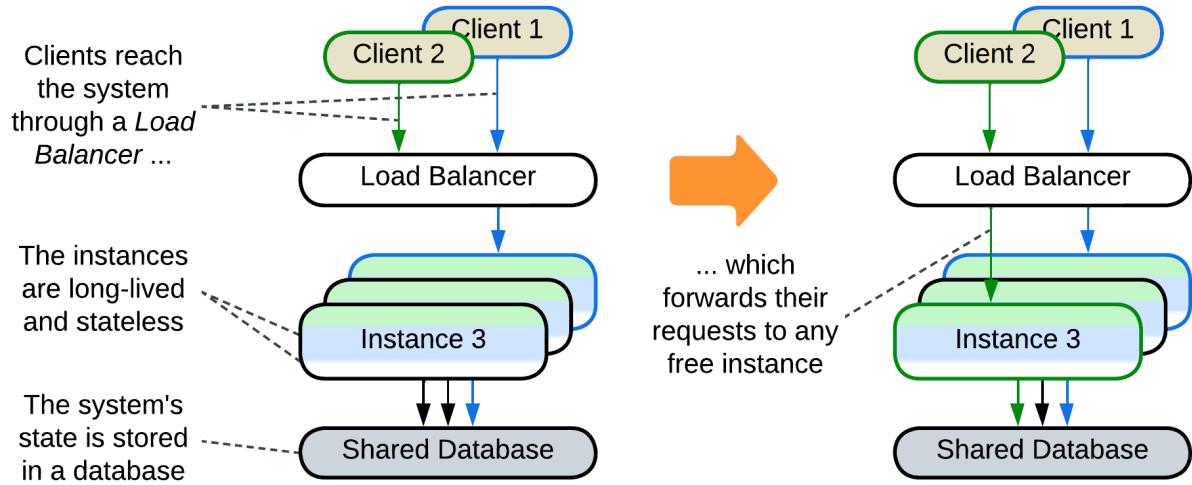
The hard part comes from the need to keep the replicas' data in sync. The ordinary way is to let the replicas talk to each other on each data update. If the communication is synchronous that may greatly slow down the processing of requests, while if it is asynchronous the system suffers data conflicts when multiple clients change the same value simultaneously. Synchronization code is quite complex, thus you will likely use a ready-made [Space-Based Architecture](#) framework instead of writing one of your own.

Another option found in the field is keeping the replicas only loosely identical. That happens when isolated cache servers make a [Caching Layer](#) [DDS]. As clients tend to send similar requests, the data inside cache is more or less the same by the law of large numbers.

And if your traffic is read-heavy, you may turn to [Polyglot Persistence](#) by segregating your replicas into the roles of a fully-functional *leader* [DDIA] and [derived, read-only followers](#). The followers serve only the read requests while the leader processes the write requests which make it update its data and broadcast the changes to all its followers. And if the leader dies, one of its followers is elected to become a new leader. As a refinement of this idea, the code of the service itself may be separated into write (*command*) and read (*query*) services ([Command Query Responsibility Segregation](#) aka CQRS).

Finally, you can mix sharding and replication to make sure that the data of each shard is replicated, either in whole among identical components [DDS] or piecemeal all over the system [DDIA]. That achieves fault tolerance for volumes of data too large to store unsharded.

Stateless: Pool, Instances, Replicated Stateless Services, Work Queue

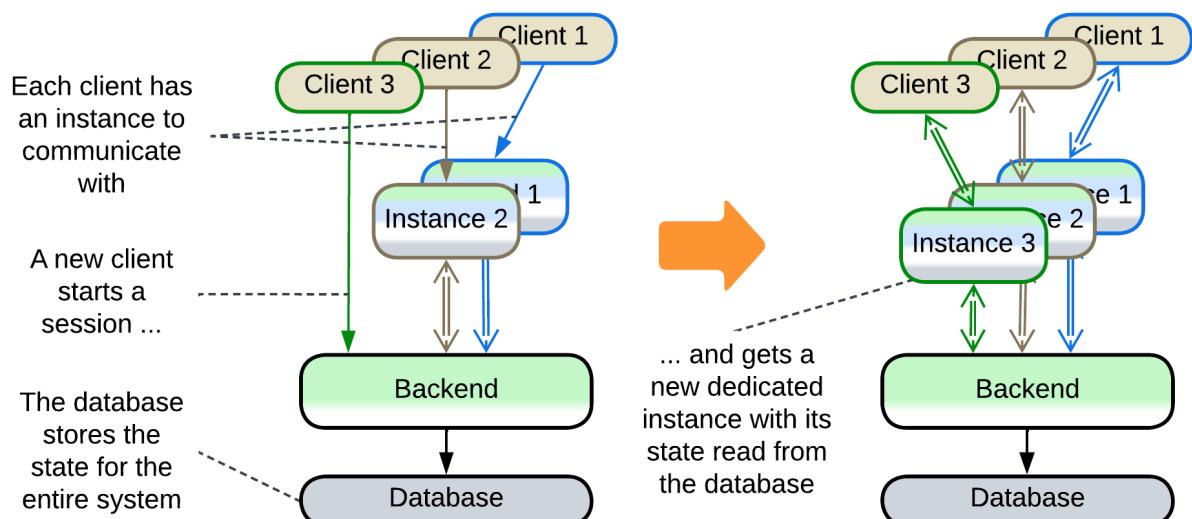


A predefined number (*pool* [[POSA3](#)]) of instances (*workers*) is created during the initialization of the system (*Work Queue* [[DDS](#)]). When the system receives a task, a [Load Balancer](#) assigns it to one of the idle instances from the pool. As soon as the instance finishes processing its task it returns to the pool. The instances don't store any state while idle, thus they are also called *Replicated Stateless Services* [[DDS](#)]. A well-known example of this pattern is [FastCGI](#).

This approach allows for rapid allocation of a worker to any incoming task, but it uses a lot of resources even when there are no requests to serve and the system may still be overwhelmed at peak load. Moreover, a [Shared Database](#) is usually involved for the sake of persistent storage, limiting the pattern's scalability.

Many cloud services implement dynamic pools, the number of instances growing and shrinking according to the overall load: if all the current instances are busy serving user requests, new instances are created and added to the pool. If some of the instances are idle for a while, they are destroyed. Dynamic pooling is often implemented through [Mesh](#), as in [Microservices](#) or [Space-Based Architecture](#).

Temporary state: Create on Demand



An instance is created for serving an incoming request and is destroyed when the request processing is finished. Upon creation it is initialized with all the client-related data to be able to interact with the client without much help from the backend. Examples include running web applications in clients' browsers and client-dedicated [actors](#) in backends of instant messengers.

This approach provides perfect elasticity and flexibility of deployment at the cost of slower session establishment and it usually relies on an external shared layer for persistence: instances of a frontend are initialized from and send their updates to a backend which itself uses a database.

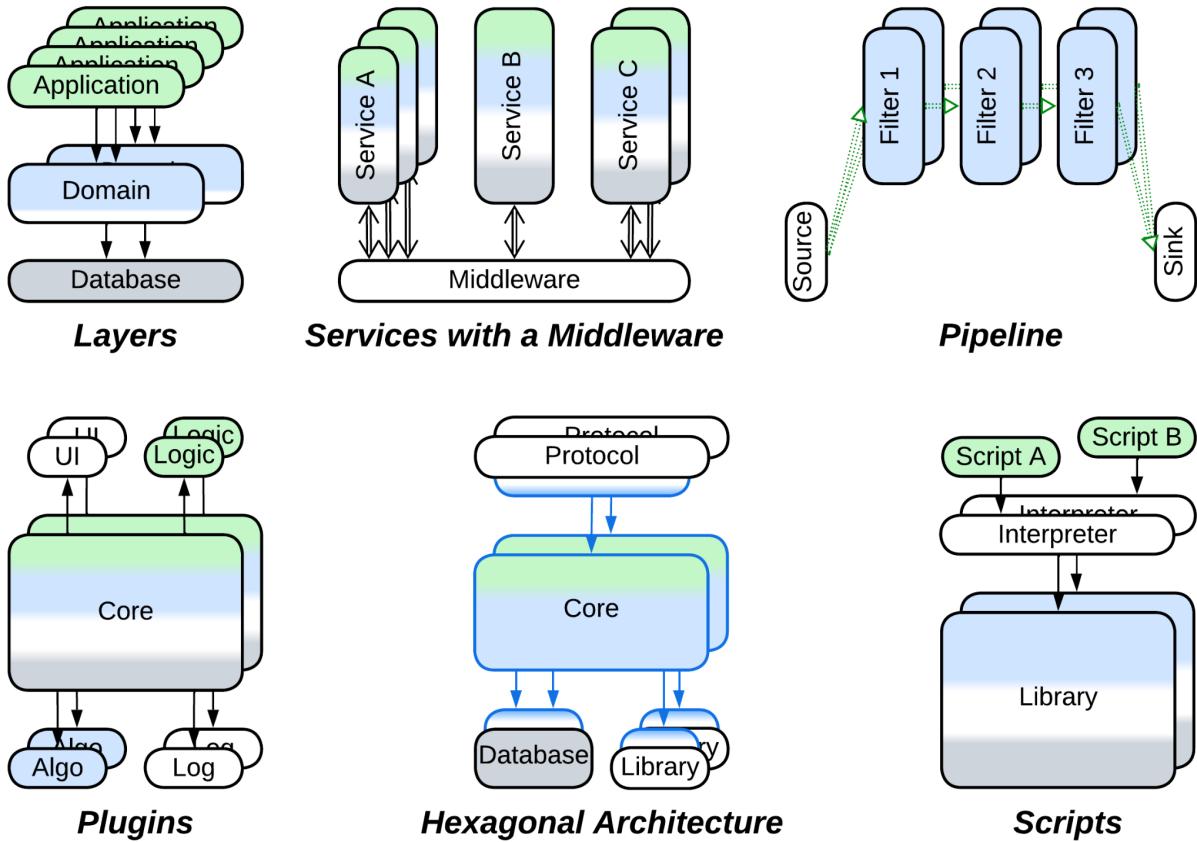
Evolutions

There are two kinds of evolutions for *Shards*: those intrinsic to the component sharded and those specific to the *Shards* pattern. All of them are summarized below while [Appendix E](#) provides more details on the second kind.

Evolutions of a sharded monolith

When *Shards* are applied to a single component, which is a [Monolith](#), the resulting (sub)system follows most of the [evolutions of Monolith](#):

- [Layers](#) allow for the parts of the system to differ in *qualities* ([forces](#)) and [deployment](#). Various third-party components can be integrated and the code becomes better structured.
- [Services](#) or [Pipeline](#) help to distribute the work among multiple teams and may decrease the project's complexity if the division yields loosely coupled components.
- [Plugins](#) and its subtypes, namely [Hexagonal Architecture](#) and [Scripts](#), make the system more adaptable.

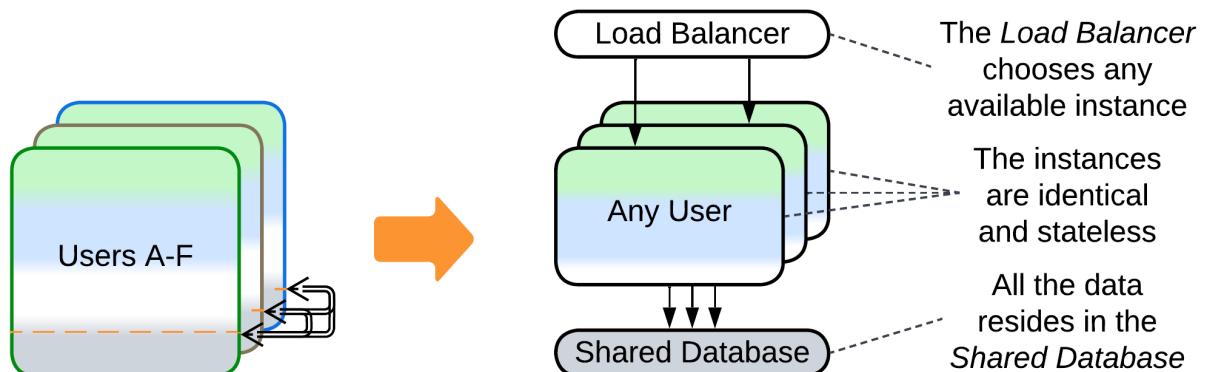


There is a benefit of such transformations which is important in the context of *Shards*: in many cases the resulting components can be scaled independently, arranging for a better resource utilization by the system (when compared to scaling a *Monolith*). However, scaling individual services usually requires a [Load Balancer](#) or [Middleware](#) to distribute requests over the scaled instances.

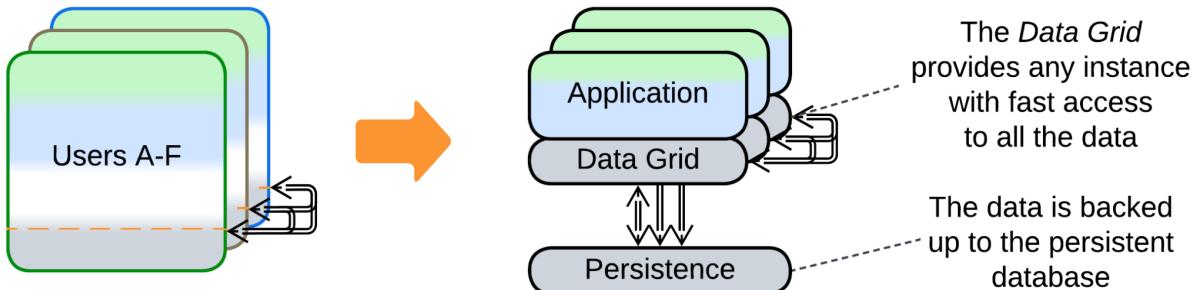
Evolutions that share data

The issue peculiar to *Shards* is that of coordinating deployed instances, especially if their data becomes coupled. The most direct solution is to let the instances access the shared data:

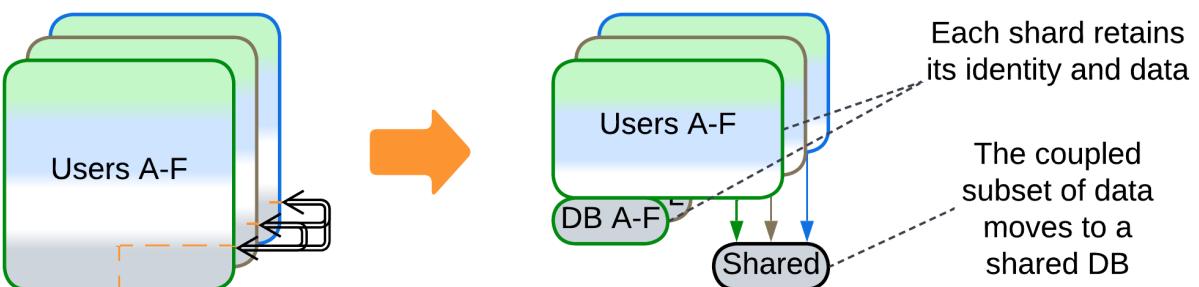
- If the whole dataset needs to be shared, it can be split into a [Shared Repository](#) layer.



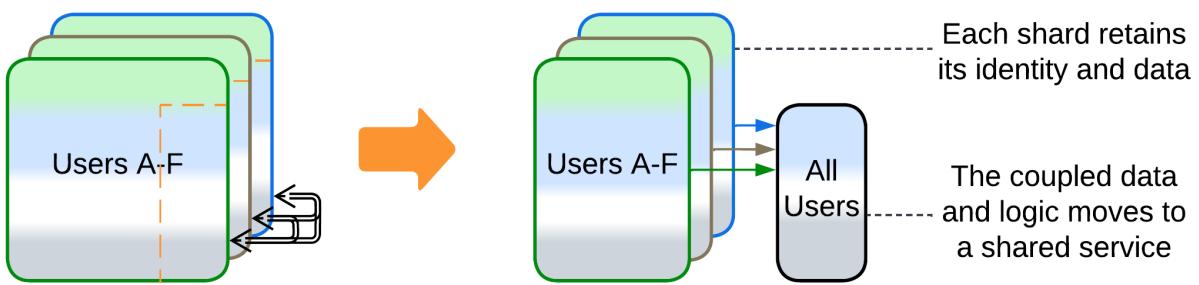
- If data collisions are tolerated, [Space-Based Architecture](#) promises low latency and dynamic scalability.



- If a part of the system's data becomes coupled, only that part can be moved to a *Shared Repository*, making each instance manage two stores of data: private and shared.



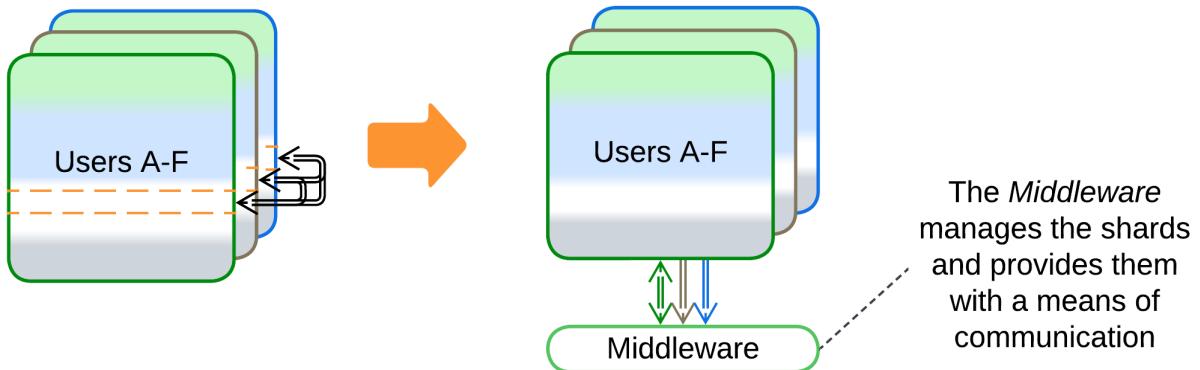
- Another possible option is to split a *service* that owns the coupled data and is always deployed as a single instance. The remaining parts of the system become coupled to that service, not to each other.



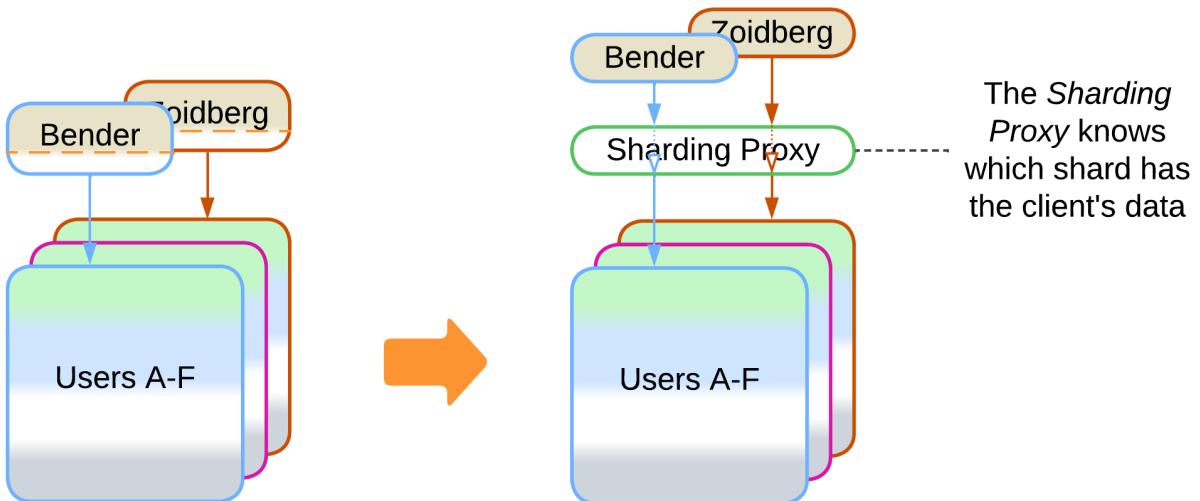
Evolutions that share logic

Other cases are better solved by extracting the logic that manipulates multiple shards:

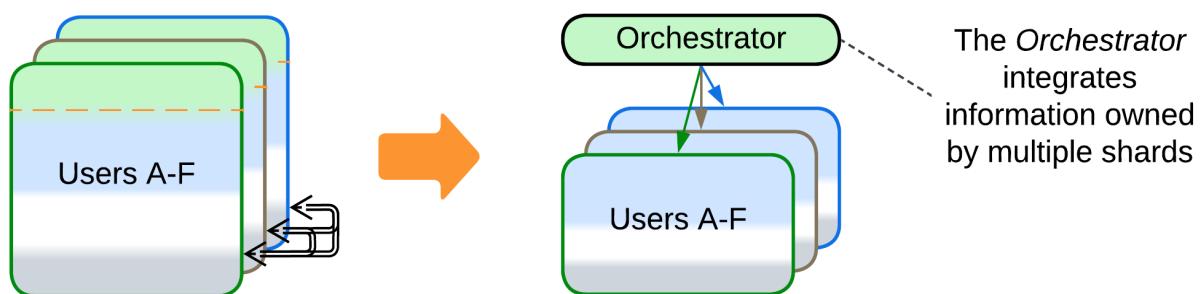
- Splitting a *service* (as discussed above) yields a component that represents both shared data and shared logic.
- Adding a *Middleware* lets the shards communicate with each other without maintaining direct connections. It also may do housekeeping: error recovery, replication, and scaling.



- A [*Sharding Proxy*](#) hides the shards from the system's clients.



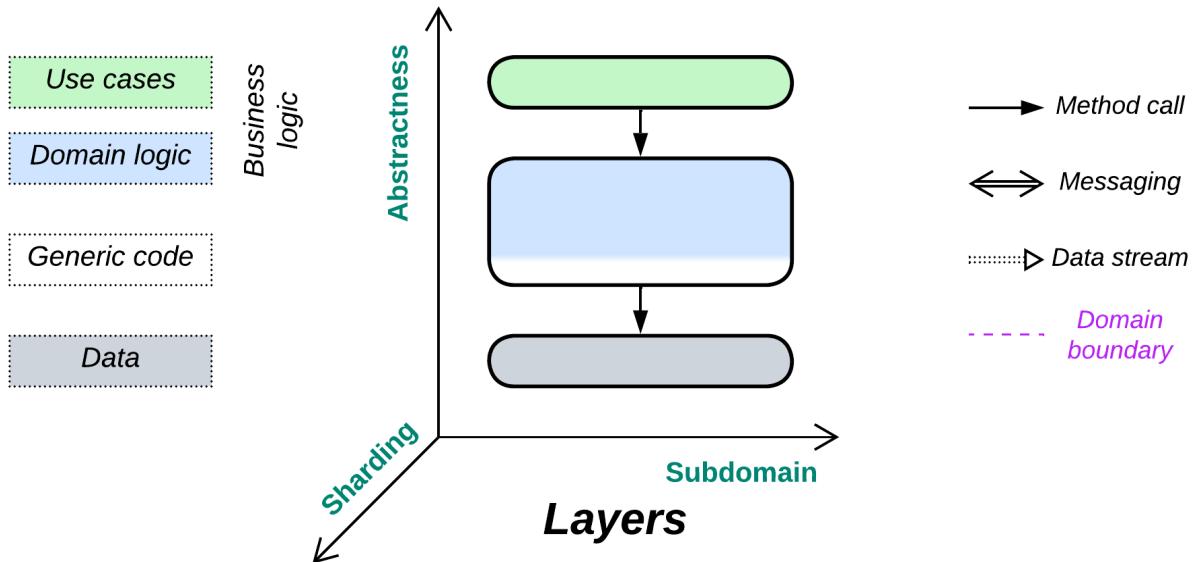
- An [*Orchestrator*](#) calls multiple shards to serve a user request. That relieves the shards of the need to coordinate their states and actions by themselves.



Summary

Shards are multiple instances of a component or subsystem which is thus made scalable and more fault tolerant. *Sharding* does not change the nature of the component it applies to and it usually relies on a [*Load Balancer*](#) or [*Middleware*](#) to manage the instances it spawns. Its main weakness appears when the *shards* need to intercommunicate, often to the end of synchronizing their data.

Layers



Yet another layer of indirection. Don't mix the business logic and implementation details.

Known as: Layers [[POSA1](#), [POSA4](#)], Layered Architecture [[SAP](#), [FSA](#), [LDDD](#)], Multitier Architecture, and N-tier Architecture [[LDDD](#)].

Variants: Open or closed, the number of layers.

By isolation:

- Synchronous layers / Layered Monolith [[FSA](#)],
- Asynchronous layers,
- A process per layer,
- Distributed tiers.

Examples:

- Domain-Driven Design (DDD) Layers [[DDD](#)],
- Three-Tier Architecture,
- Embedded Systems.

Structure: A component per level of abstractness.

Type: Main, implementation.

Benefits	Drawbacks
Rapid start for development	Quickly deteriorates as the project grows
Easy debugging	Hard to develop with more than a few teams
Good performance	Does not solve force conflicts between subdomains

Development teams may specialize

Business logic is encapsulated

Allows resolution of conflicting forces

Deployment to dedicated hardware

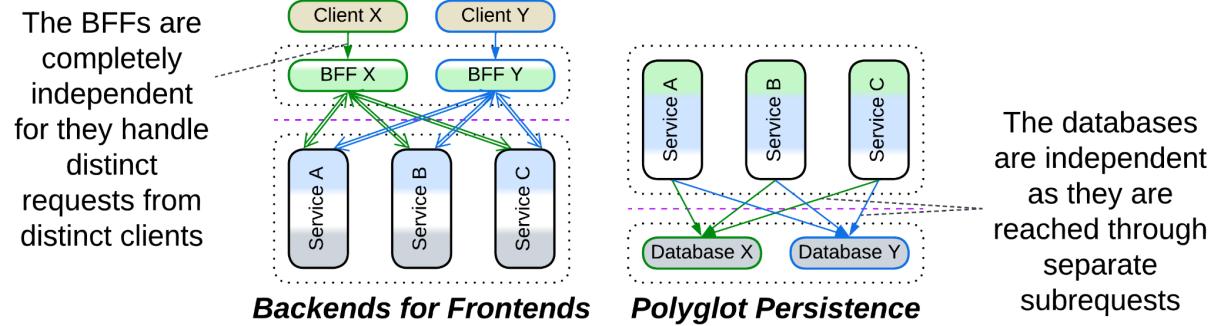
Layers with no business logic are reusable

References: [[POSA1](#)] and [[FSA](#)] discuss layered software in depth; [[DDD](#)] promotes the layered style; most of the architectures in Herberto Graça's [Software Architecture Chronicles](#) are layered. The Wiki has a reasonably [good article](#).

Layering a system creates interfaces between its levels of abstractness (high-level use cases, lower-level domain logic, infrastructure) while also retaining monolithic cohesiveness within each of the levels. That allows both for easy debugging inside each individual layer (no need to jump into another programming language or re-attach the debugger to a remote server) and enough flexibility to have a dedicated development team, tools, deployment, and scaling policies for each. Though layered code is slightly better than that of [Monolith](#), thanks to the separation of concerns, one of the upper (business logic) layers may nonetheless grow too large for efficient development.

Splitting a system into layers tends to resolve conflicts of forces between its abstract and optimized parts: the top-level business logic changes rapidly and does not require much optimization (as its methods are called infrequently), thus it can be written in a high-level programming language. In contrast, infrastructure, which is called thousands of times per second, has stable workflows but must be thoroughly optimized and extremely well tested.

Many patterns have one or more of their layers split by subdomain, resulting in a layer of services. That causes no penalties as long as the services are completely independent (the original layer had zero coupling between its subdomains), which happens if each of them deals with a separate subset of requests (as in [Backends for Frontends](#)) or is choreographed by an upper layer (as in [Polyglot Persistence](#), [Hexagonal Architecture](#) or [Hierarchy](#)) which boils down to the same “separate subset of subrequests” under the hood. However, if the services which form a layer need to intercommunicate, you immediately get a whole set of troubles with debugging, sharing data, and performance characteristic of the [Services](#) architecture.



Thanks to its substantial benefits and minor drawbacks, and the many evolutions it supports, *Layers* became the default architecture for starting new projects.

Performance

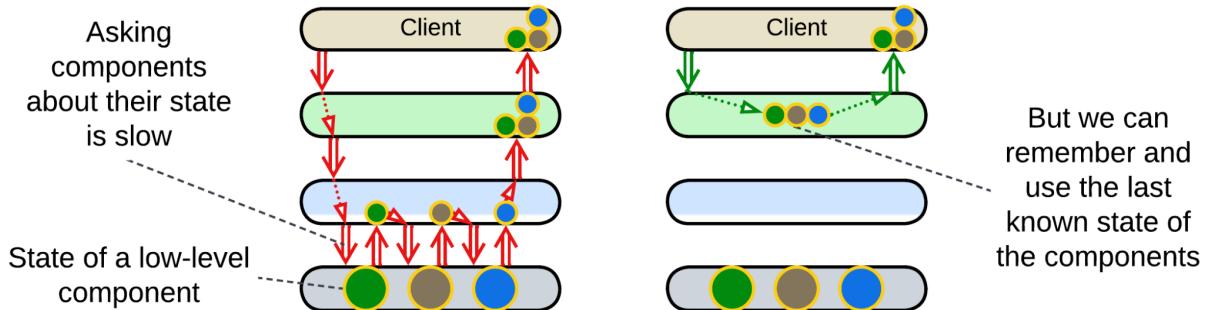
The performance of a layered system is shaped by two factors:

- Communication between layers is slower than within a layer. Components of a layer may access each other’s data directly, while accessing another layer involves data transformation (as interfaces tend to operate generic data structures), serialization, and often IPC or networking.
- The frequency and granularity of events or actions increases as we move from the upper more abstract layers to lower-level components that interface an OS or hardware.

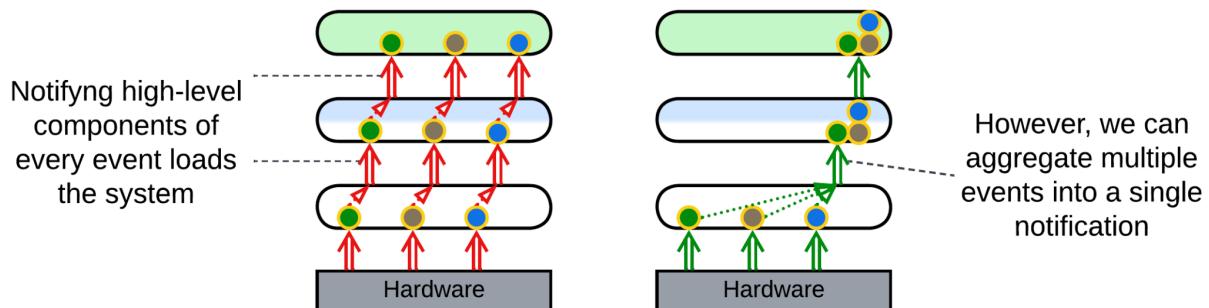
An ideal component should be replaceable and reusable. As soon as a component exposes details of its implementation, such as workflows or data types, in its interface, it becomes incompatible with other possible implementations, and its interface may even see major changes as the internals of the component evolve.

Therefore, well-behaving components tend to have their interfaces written in most generic terms, which requires inputs to be transformed to their internal formats and thus penalizes performance.

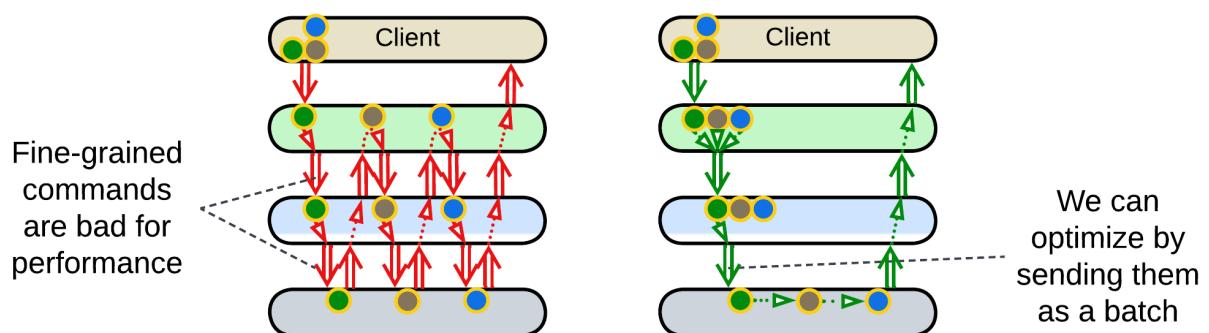
There is a number of optimizations to skip interlayer calls:



Caching: an upper layer tends to *model* (cache last known state of) the layers below it. This way it can behave as if it knew the state of the whole system without querying the actual state from the hardware below all the layers. Such an approach is universal for [control software](#). For example, a network monitoring suite shows you the last known state of all the components it observes without actually querying them – it is subscribed to notifications and remembers what each device has previously reported.

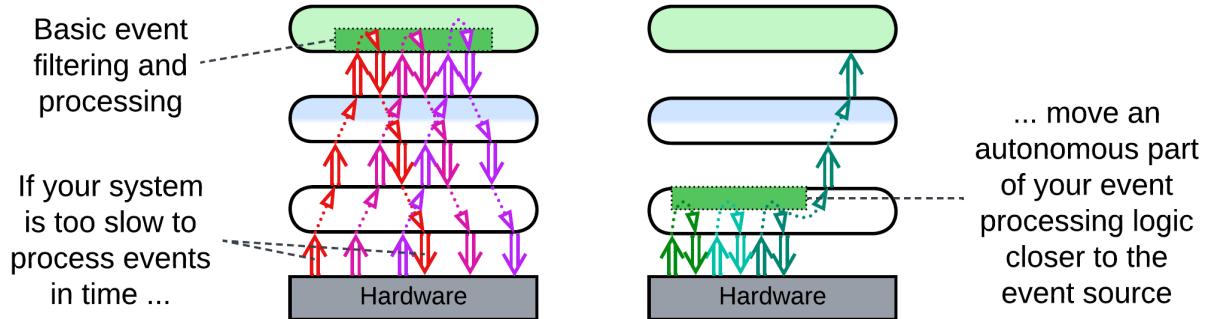


Aggregation: a lower layer collects multiple events before notifying the layer above it to avoid being overly chatty. An example is an [IIoT](#) field gateway that collects data from all the sensors in the building and sends it in a single report to the server. Or consider a data transfer over a network where a low-level driver collects multiple data packets that come from the hardware and sends an acknowledgement for each of them while waiting for a datagram or file transfer to complete. It notifies its client only once when all the data has been collected and its integrity confirmed.



Batching: an upper layer forms a queue of commands and sends it as a single job to the layer below it. This takes place in drivers for complex low-level hardware, like printers, or in database access as *stored procedures*. [\[POSA4\]](#) describes the approach as *Combined*

Method, *Enumeration Method* and *Batch Method* patterns. Programming languages and frameworks may implement *foreach* and *map/reduce* which allow for a single command to operate on multiple pieces of data.



Strategy injection: an upper layer installs an event handler (hook) into the lower layer. The goal is for the hook to do basic pre-processing, filtering, aggregation, and decision making to process the majority of events autonomously while escalating to the upper layer in exceptional or important cases. That may help in such time-critical domains as high-frequency trading.

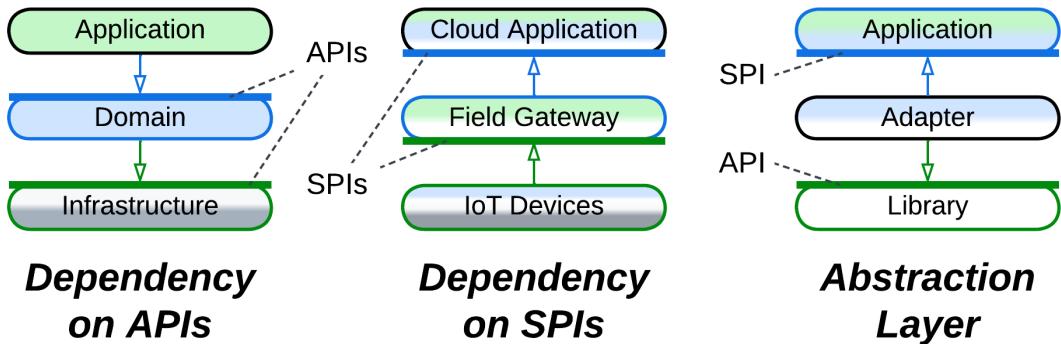
Layers can be scaled independently, as [exemplified](#) by common web applications that comprise a highly scalable and resource-consuming frontend, somewhat scalable backend and unscalable data layer. Another example is an OS (lower layer) that runs multiple user applications (upper layer).

Dependencies

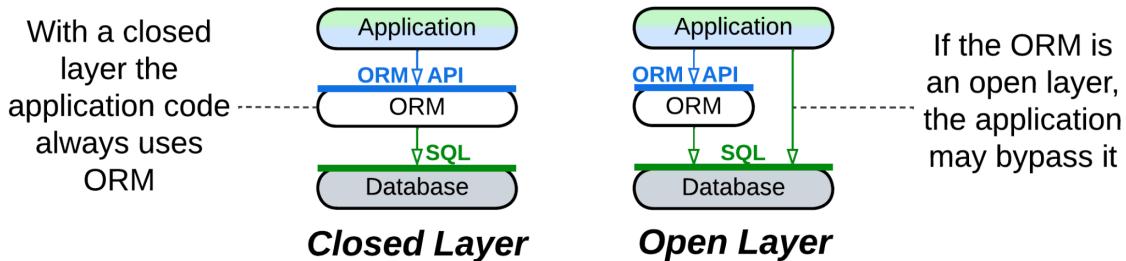
Usually an upper layer depends on the *API* (application programming interface) of the layer directly below it. That makes sense as the lower the layer is, the more stable it tends to be: a user-facing software gets updated on a daily or weekly basis while hardware drivers may not change for years. As every update of a component may destabilize other components that depend on it, it is much more preferable for a quickly evolving component to depend on others instead of the other way round.

Some domains, including embedded systems and telecom, require their lower layers to be polymorphic as they deal with varied hardware or communication protocols. In that case an upper layer (e.g. OS kernel) defines a *service provider interface (SPI)* which is implemented by every variant of the lower layer (e.g. a device driver). That allows for a single implementation of the upper layer to be interoperable with any subclass of the lower layer. Such an approach enables [Plugins](#), [Microkernel](#), and [Hexagonal Architecture](#).

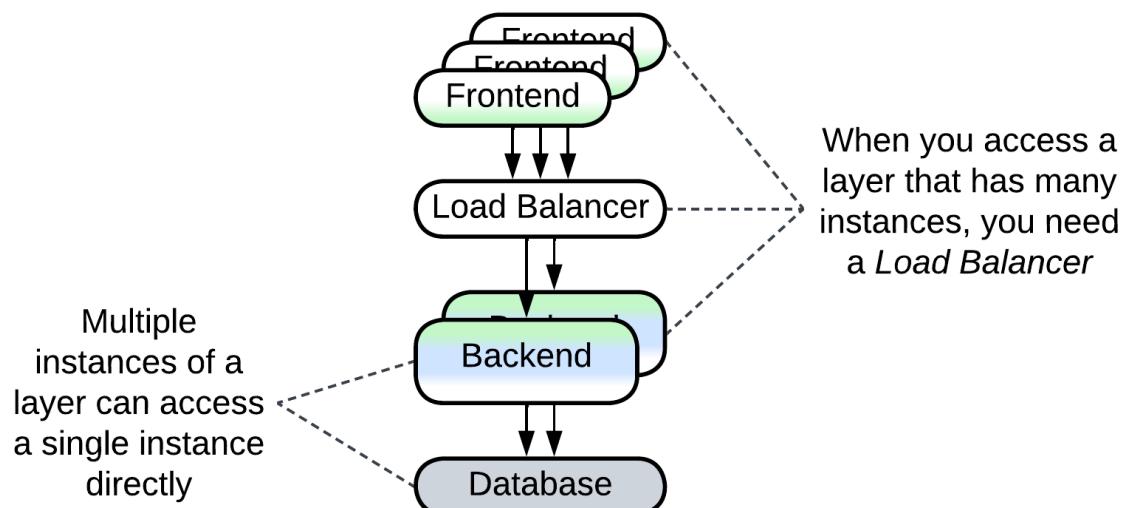
There may also be an [Adapter](#) layer between your system's SPI and an external API. It is called *Anticorruption Layer* [[DDD](#)], *Database Abstraction Layer* / *Database Access Layer* [[POSA4](#)] / *Data Mapper* [[PEAA](#)], *OS Abstraction Layer* or *Platform Abstraction Layer* / *Hardware Abstraction Layer*, depending on what kind of component it adapts.



A layer can be *closed (strict)* or *open (relaxed)*. A layer above a closed layer depends only on the closed layer right below it – it does not see through it. Conversely, a layer above an open layer depends on both the open layer and the layer below it. The open layer is transparent. That helps keep a layer which encapsulates one or two subdomains small: if such a layer were closed, it would have to copy much of the interface of the layer below it just to pass the incoming requests which it does not handle through to the layer below. The optimization of the open layer has a cost: the team that works on the layer above an open layer needs to learn two APIs which may have incompatible terminology.



If you ever need to *scale* (run multiple instances of) a layer, you may notice that a layer which sends requests naturally supports multiple instances, with the instance address being appended to each request so that its destination layer knows where to send the response. On the other hand, if there are multiple instances of a layer you call into, you need a kind of [Load Balancer](#) to dispatch requests among the instances.



Applicability

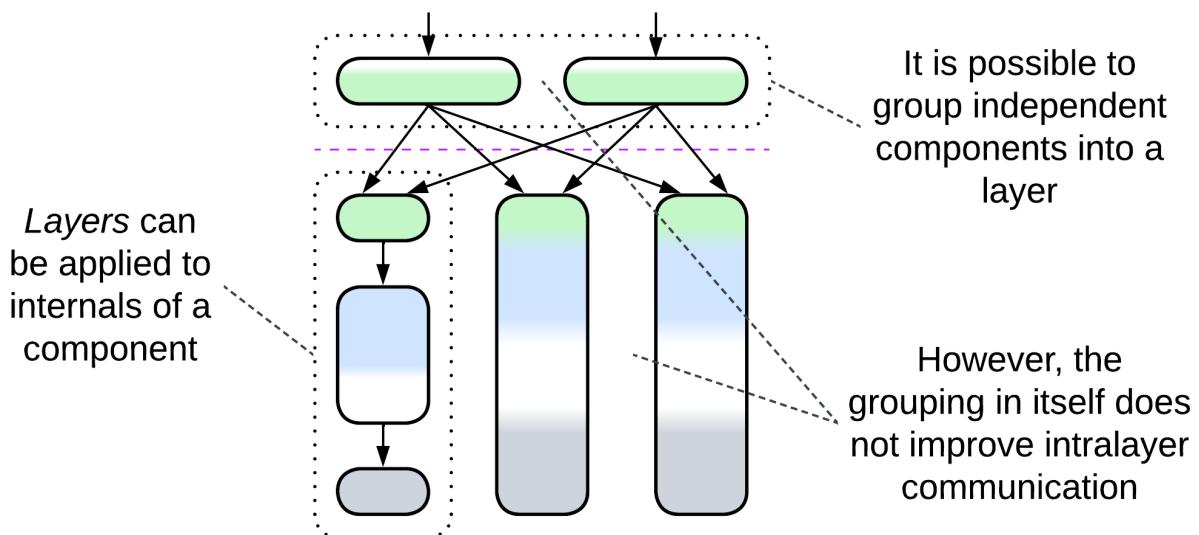
Layers are good for:

- *Small and medium-sized projects.* Separating the business logic from the low-level code should be enough to work comfortably on codebases below 100 000 lines in size.
- *Specialized teams.* You can have a team per layer: some people, who are proficient in optimization, work on the highly loaded infrastructure, while others talk to the customers and write the business logic.
- *Deployment to a specific hardware.* Frontend instances run on client devices, a backend needs much RAM, the data layer demands a large HDD and security. There is no way to unite them into a single generic component.
- *Flexible scaling.* It is common to have hundreds or thousands of frontend instances being served by several backend processes that use a single database.
- *Real-time systems.* Hardware components and network events often need the software to respond within a set time limit. This is achievable by separating the time-critical code from normal priority calculations. See [Hexagonal Architecture](#) and [Microkernel](#) for improved solutions.

Layers are bad for:

- *Large projects.* You are still going to enter *monolithic hell* [[MP](#)] if you reach 1 000 000 lines of code.
- *Low-latency decision making.* If your business logic needs to be applied in real time, you cannot tolerate the extra latency caused by the interlayer communication.

Relations



Layers:

- Can be applied to the internals of any module, for example, layering [Services](#) results in [Layered Services](#).
- Can be altered by [Plugins](#) or extended with a [Proxy](#), [Orchestrator](#), and/or [Shared Repository](#) to form an extra layer.
- Can be implemented by [Services](#) yielding layers of services present in [Service-Oriented Architecture](#), [Backends for Frontends](#), or [Polyglot Persistence](#).

- May be closely related to [Hexagonal Architecture](#) or the derived [Separated Presentation](#).
- A layer often serves as a [Proxy](#), [Orchestrator](#), and/or [\(Shared\) Repository](#).

Variants by isolation

There are [several grades](#) of layer isolation between unstructured [Monolith](#) and distributed *Tiers*. All of them are widely used in practice: each step adds its specific benefits and drawbacks to those of the previous stages until at some point it makes more sense to reject the next deal because its cons are too inconvenient for you.

Synchronous layers, Layered Monolith

First you separate the high-level logic from low-level implementation details. Then draw interfaces between them. The layers will still call each other directly, but at least the code has been sorted out into some kind of structure, and you can now employ two or three dedicated teams, one per layer. The cost is quite low – it is that the newly created interfaces will stand in the way of aggressive performance optimization.

Benefits	Drawbacks
Structured code	Lost opportunities for optimization
Two or three teams	

Asynchronous layers

For the next step you may decide to take will be to isolate the layers' execution threads and data. The layers will communicate only through in-process messages, which are slower than direct calls and harder to debug, but now each layer can run at its own pace – a must for interactive systems.

Benefits	Drawbacks
Structured code	No opportunities for optimization
Two or three teams	Some troubles with debugging
The layers may differ in latency	

A process per layer

Next, you may run each layer in a separate process. You have to devise an efficient means of communication between them, but now the layers may differ in technologies, security, frequency of deployment, and even stability – the crash of one layer does not impact any of the others. Moreover, you may scale each layer to make good use of the available CPU cores. However, you will pay through even harder debugging, lower performance, and you will have to take care of error recovery, because if one of the components crashes, the others are likely to remain in an inconsistent state.

Benefits	Drawbacks
Structured code	No opportunities for optimization
Two or three teams	Troublesome debugging
The layers may differ in latency	Some performance penalty
The layers may differ in technologies	Error recovery must be addressed
The layers are deployed independently	

Software security isolation
Software fault isolation
Limited scalability

Distributed tiers

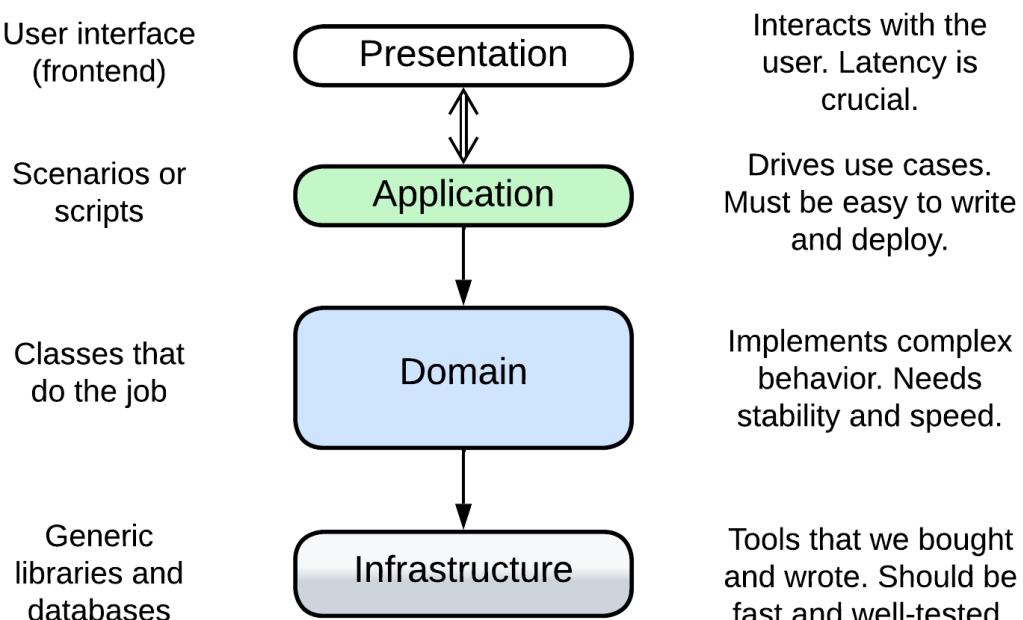
Finally, you may separate the hardware which the processes run on – going all out for distribution. This allows you to fine-tune the system configuration, run parts of the system close to its clients, and physically isolate the most secure components, with your scalability limited only by your budget. The price is paid in latency and debugging experience.

Benefits	Drawbacks
Structured code	No opportunities for optimization
Two or three teams	Even worse debugging
The layers may differ in latency	Definite performance penalty
The layers may differ in technologies	Error recovery must be addressed
The layers are deployed independently	
Full security isolation	
Full fault isolation	
Full scalability	
Layers vary in hardware setup	
Deployment close to clients	

Examples

The notion of layering seems to be so natural to our minds that most known architectures are layered. Not surprisingly, there are several approaches to assigning functionality to and naming the layers:

Domain-Driven Design (DDD) Layers



[DDD] recognizes four layers with the upper layers closer to the user:

- *Presentation (User Interface)* – the user-facing component (frontend, UI). It should be highly responsive to the user's input. See [Separated Presentation](#).
- *Application (Integration, Service)* – the high-level scenarios which build upon the API of the *domain* layer. It should be easy to change and to deploy. See [Orchestrator](#).
- *Domain (Model, Business Rules)* – the bulk of the mid- and low-level business logic. It should usually be well-tested and performant.
- *Infrastructure (Utility, Data Access)* – the utility components devoid of business logic. Their stability and performance is business-critical but updates to their code are rare.

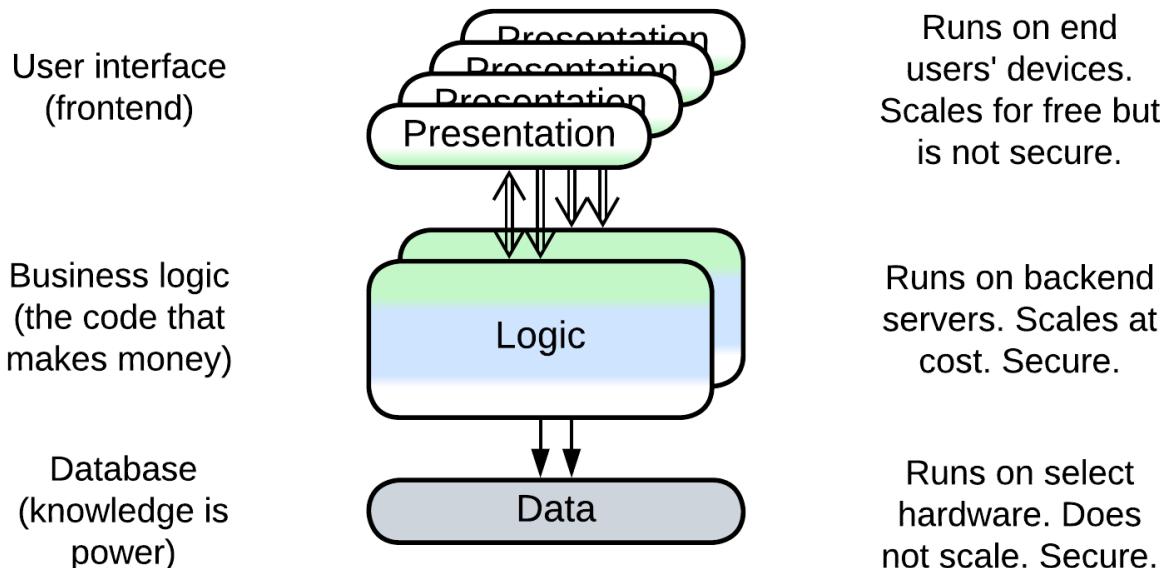
For example, an online banking system comprises:

- the presentation layer which is its frontend;
- the application layer which implements sequences of steps for payment, card to card transfer, and viewing a client's history of transactions;
- the domain layer contains classes for various kinds of cards and accounts;
- the infrastructure layer with a database and libraries for encryption and interbank communication.

However in practice you are much more likely to encounter the derived [DDD-style Hexagonal Architecture](#) than the original *DDD Layers*.

We will often use the DDD naming convention while describing more complex architectures, some of which have all the four layers, while others merge several layers together or omit the frontend and concentrate on the components that contain the business logic.

Three-Tier Architecture



Here the focus lies with the distribution of the components over heterogeneous hardware (*Tiers*):

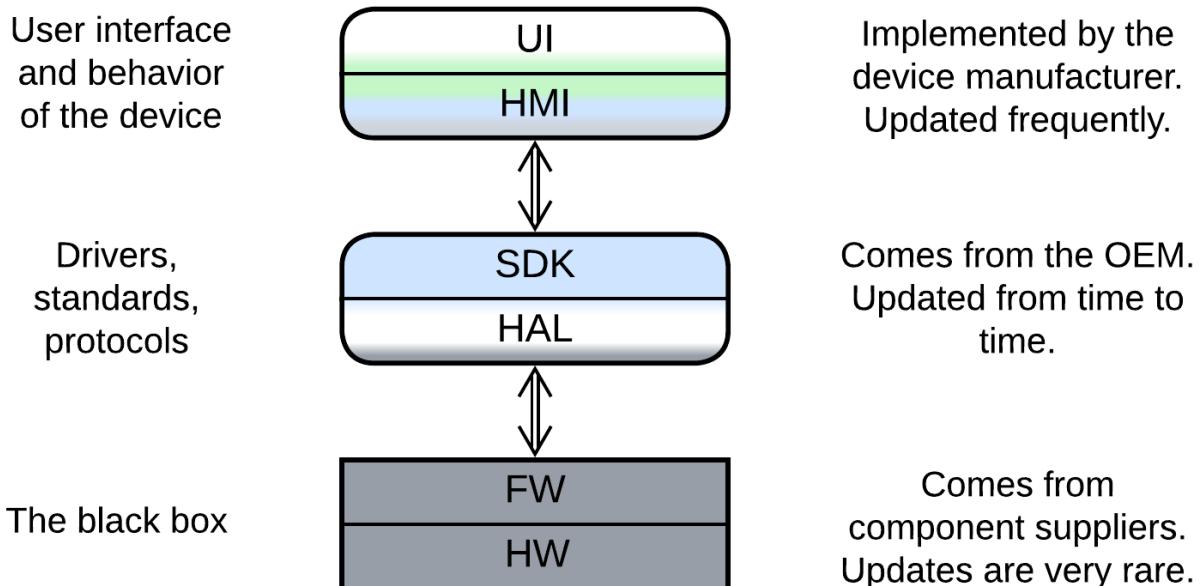
- *Presentation (Frontend)* tier – a user-facing application which runs on a user's hardware. It is very scalable and responsive, but insecure.
- *Logic (Backend)* tier – the business logic which is deployed on the service provider's side. Its scalability is limited mostly by the funding committed, security is good but latency is high.

- *Data (Database)* tier – a service provider's database which runs on a dedicated server. It is not scalable but is very secure.

In this case the division into layers resolves the conflict between scalability, latency, security, and cost as discussed in detail in the [chapter on distribution](#).

Tiers don't map to *Layers* directly. For example, a protocol support library which is used for communication between services belongs to the lowest (infrastructure) layer but to the middle (backend) tier. The discrepancy is rooted in different natures ([views of the 4+1 model](#)) of the patterns in question: *Layers* show the logical composition of the system while *Tiers* deal with its physical structure (deployment).

Embedded systems



Bare metal and micro-OS systems which run on low-end chips use a different terminology, which is not unified across domains. A generic example involves:

- *Presentation* – a UI engine used by the *HMI*. It may be a third-party library or come as a part of the *SDK*.
- *Human-Machine Interface (HMI aka MMI)* – the UI and high-level business logic for user scenarios, written by a [value-added reseller](#).
- *Software Development Kit (SDK)* – the mid-level business logic and device drivers, written by the [original equipment manufacturer](#).
- [*Hardware Abstraction Layer \(HAL\)*](#) – the low-level code that abstracts hardware registers to enable code reuse between hardware platforms.
- *Firmware of Hardware Components* – usually closed-source binary pre-programmed into chips by chipmakers.
- *Hardware itself*.

It is of note that in this approach the layers form strongly coupled pairs. Each pair is implemented by a separate party of the supply chain, which is an extra force that shapes the system into layers.

An example of such a system can be found in an old mobile phone or a digital camera.

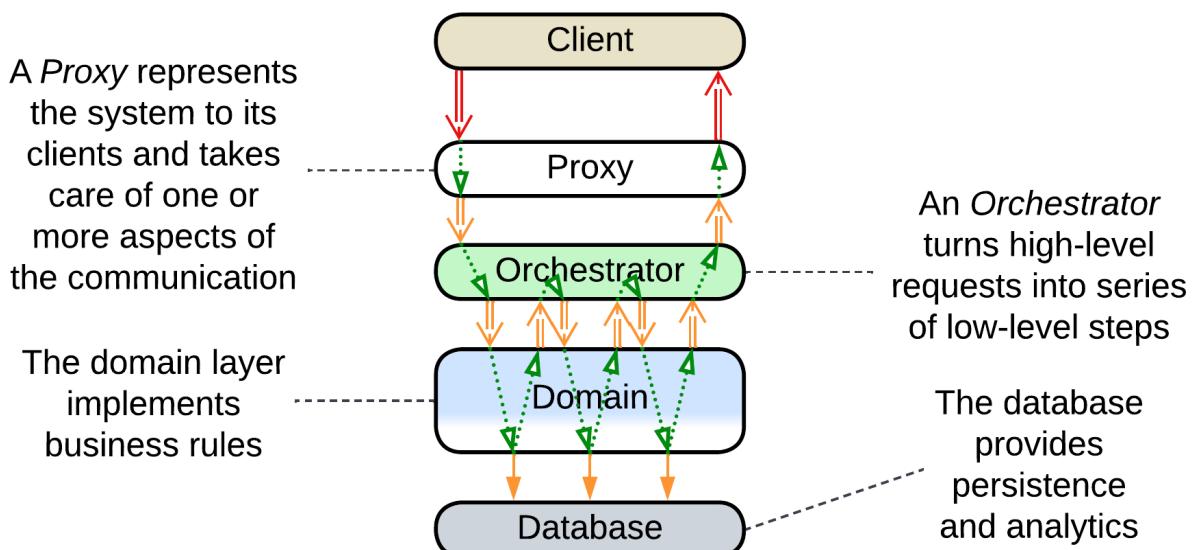
Evolutions

Layers are not without drawbacks which may force your system to evolve. A summary of such evolutions is given below while more details can be found in [Appendix E](#).

Evolutions that make more layers

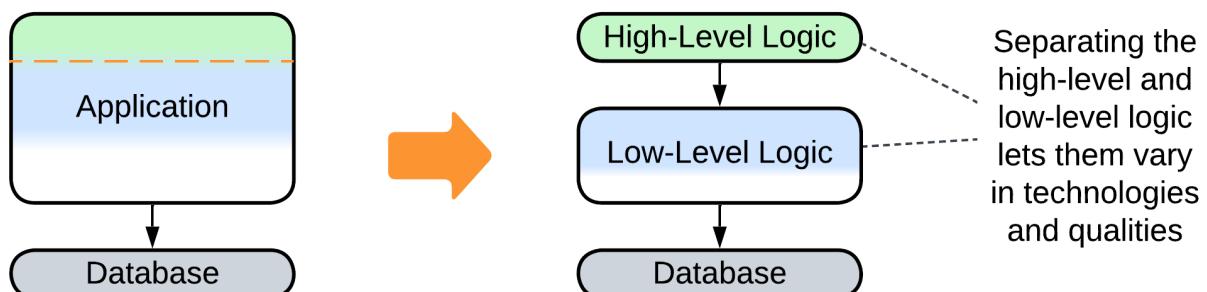
Not all the layered architectures are equally layered. A [Monolith](#) with a [Proxy](#) or database has already stepped into the realm of *Layers* but is far from reaping all its benefits. Such a system may continue its course in a few ways that were previously [discussed for Monolith](#):

- Employing a *database* (if you don't use one) lets you rely on a thoroughly optimized state-of-the-art subsystem for data processing and storage.
- [Proxies](#) are similarly reusable generic modules to be added at will.
- Implementing an [Orchestrator](#) on top of your system may improve programming experience and runtime performance for your clients.



It is also common to:

- Have the business logic divided into two layers.

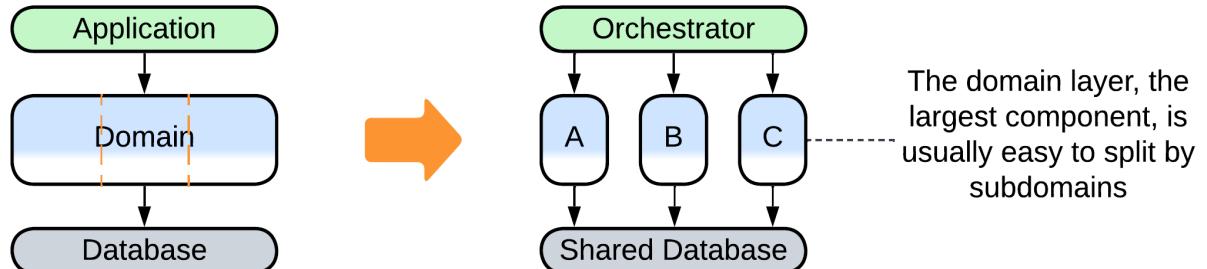


Evolutions that help large projects

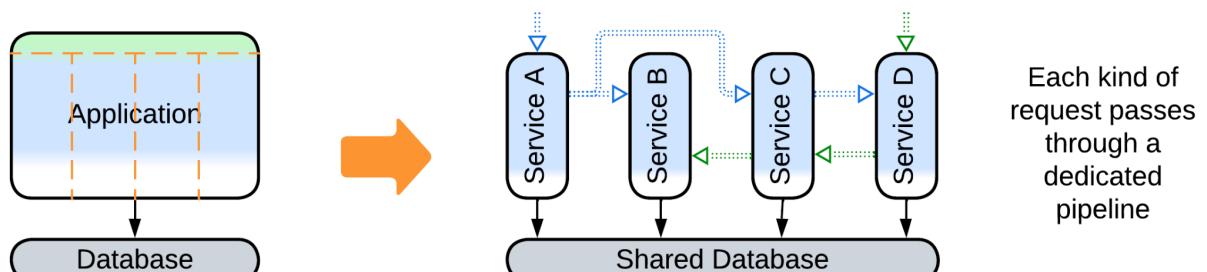
The main drawback (and benefit as well) of *Layers* is that much or all of the business logic is kept together in one or two components. That allows for easy debugging and fast development in the initial stages of the project but slows down and complicates work as the project grows in size [[MP](#)]. The only way for a growing project to survive and continue evolving at a reasonable speed is to divide its business logic into several smaller, [thus less](#)

[complex](#), components that match subdomains (*bounded contexts* [[DDD](#)]). There are several options for such a change, with their applicability depending on the domain:

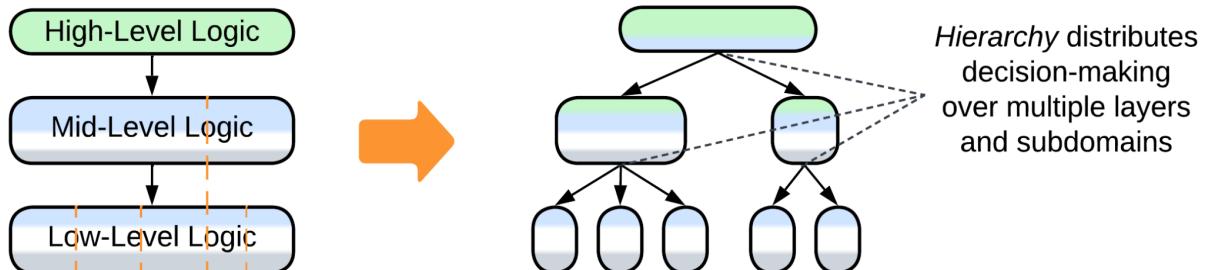
- The middle layer with the main business logic can be divided into [Services](#) leaving the upper [Orchestrator](#) and lower [database](#) layers intact for future evolutions.



- Sometimes the business logic can be represented as a set of directed graphs which is known as [Event-Driven Architecture](#).



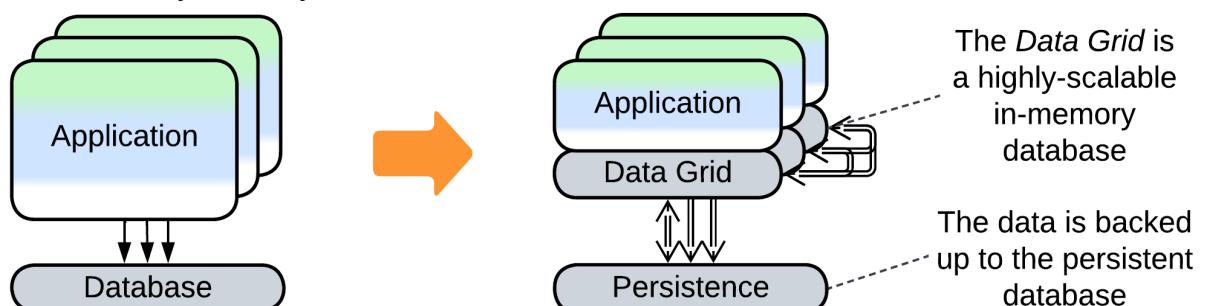
- If you are lucky, your domain makes a [Top-Down Hierarchy](#).



Evolutions that improve performance

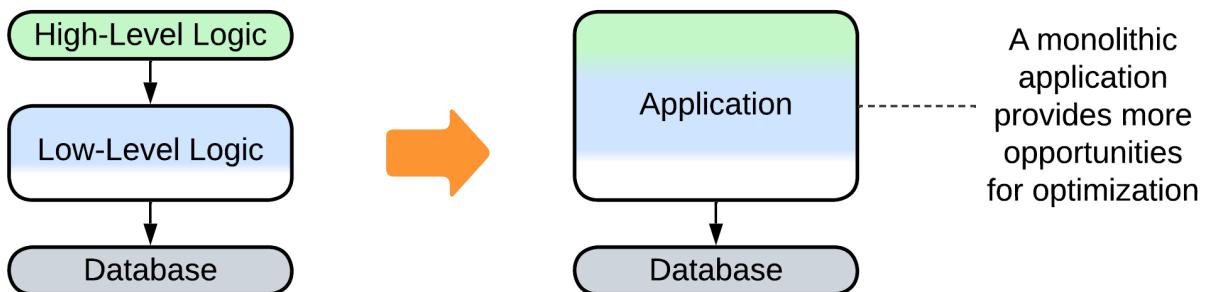
There are several ways to improve the performance of a layered system. One we have [already discussed for Shards](#):

- [Space-Based Architecture](#) co-locates the database and business logic and scales both dynamically.

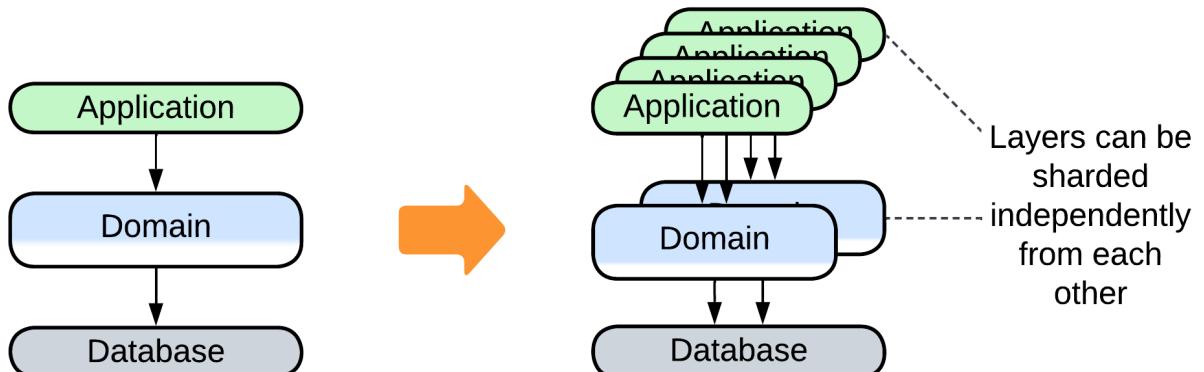


Others are new:

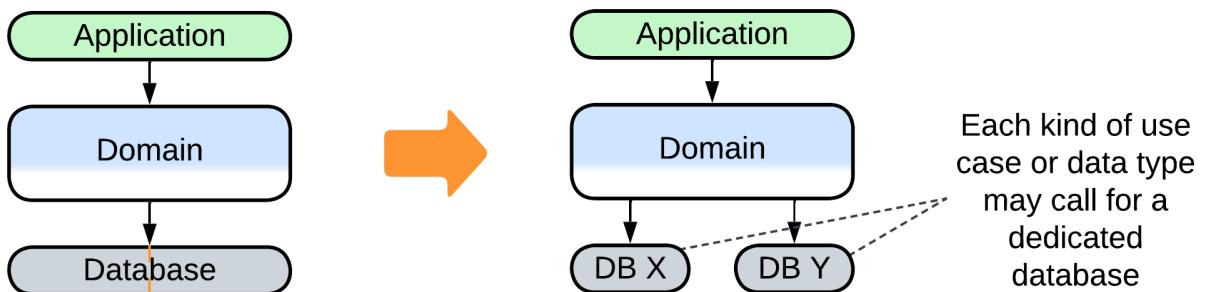
- Merging several layers improves latency by eliminating the communication overhead.



- [Scaling](#) some of the layers may improve throughput.



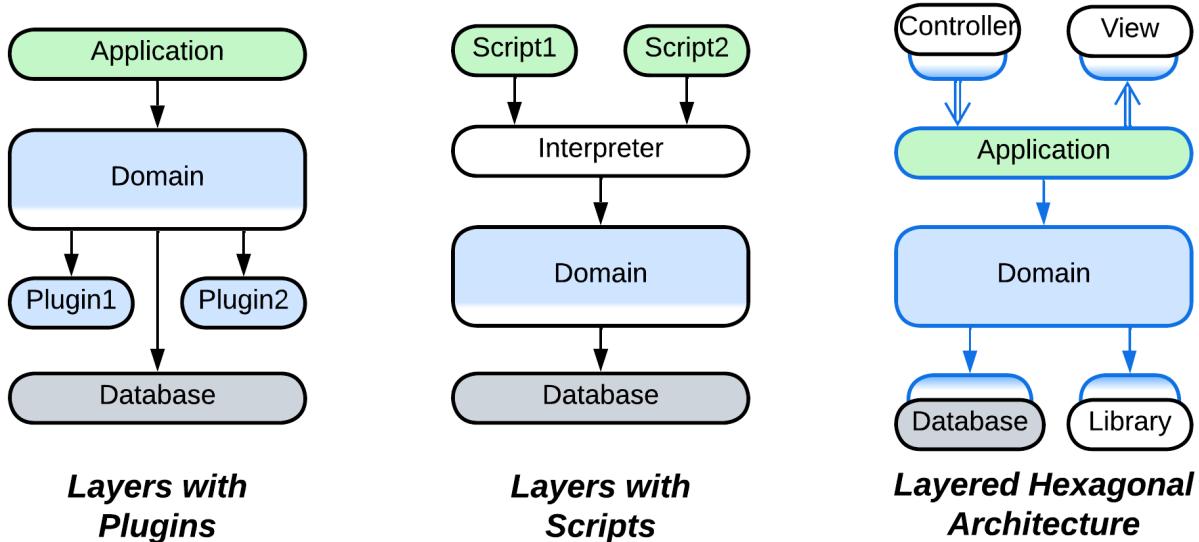
- [Polyglot Persistence](#) is the name for using multiple specialized databases.



Evolutions to gain flexibility

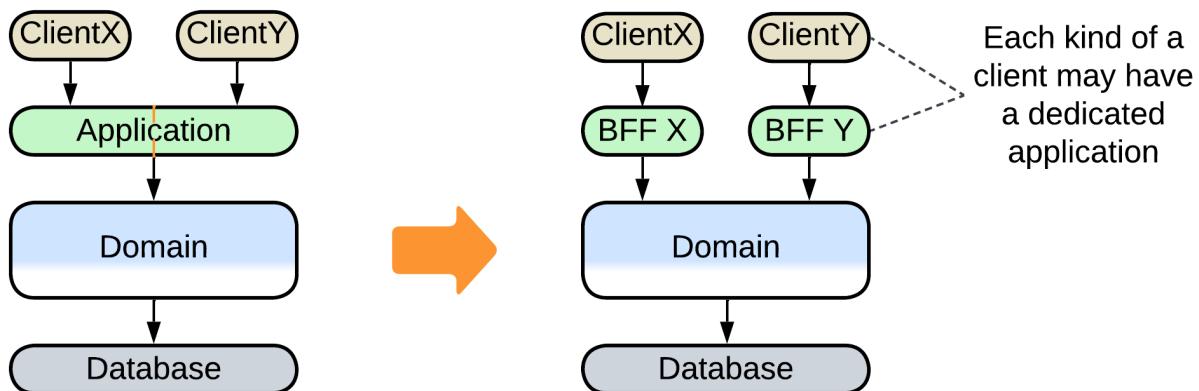
The last group of evolutions to consider is about making the system more adaptable. We have already discussed the following [evolutions for Monolith](#):

- The behavior of the system may be modified with [Plugins](#).
- [Hexagonal Architecture](#) allows for abstracting the business logic from the technologies used in the project.
- [Scripts](#) allow for customization of the system's logic on a per client basis.



There is one new evolution which modifies the upper (*orchestration*) layer:

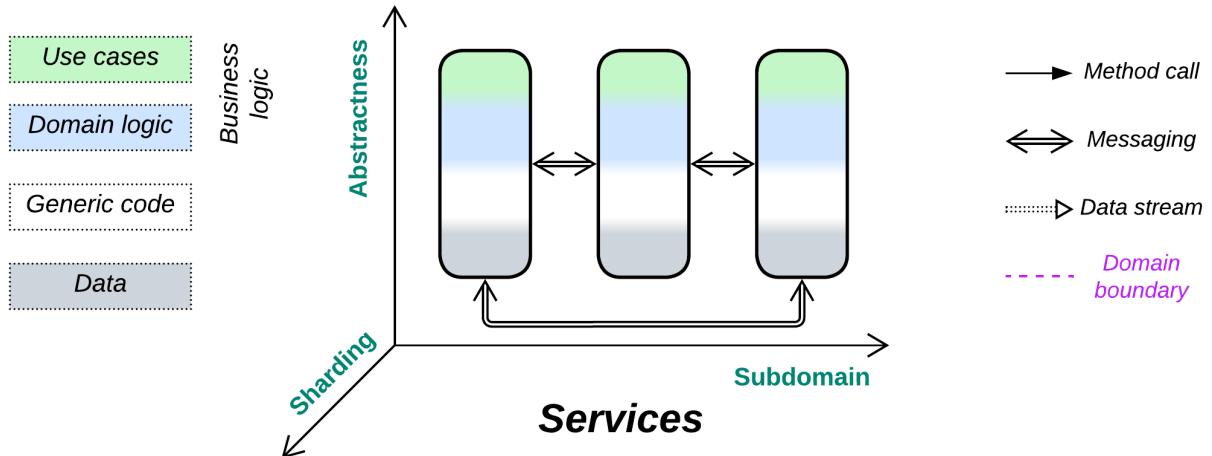
- The [orchestration layer](#) may be split into [Backends for Frontends](#) to match the individual needs of several kinds of clients.



Summary

Layered architecture separates the high-level logic from the low-level details. It is superior for medium-sized projects as it supports rapid development by two or three teams, is flexible enough to resolve conflicting forces, and provides many options for further evolution, which will come in handy when the project grows in size and complexity.

Services



Divide and conquer. Gain flexibility through decoupling subdomains.

Known as: Services, Domain Services [[FSA](#)] and [[SAHP](#)], but not DDD.

Variants: [Pipeline](#) has a dedicated chapter. Many modifications are listed in the [Evolutions section](#).

By isolation:

- Synchronous modules: [Modular Monolith](#) [[FSA](#)] (Modulith),
- Asynchronous modules: [Modular Monolith](#) (Modulith) / [Embedded Actors](#),
- Multiple processes,
- Distributed runtime: [Function as a Service](#) (FaaS) (including [Nanoservices](#)) / [Backend Actors](#),
- Distributed services: Service-Based Architecture [[FSA](#)] / Space-Based Architecture [[FSA](#)] / Microservices.

By communication:

- Direct method calls,
- [RPCs](#) and commands (request/confirm pairs),
- Notifications (pub/sub) and shared data,
- (inexact) No communication.

By size:

- Whole subdomain: Domain Services [[FSA](#)],
- Part of a subdomain: Microservices,
- Class-like: [Actors](#),
- Single function: [FaaS](#) [[DDS](#)] / [Nanoservices](#).

By internal structure:

- Monolithic service,
- Layered service,
- Hexagonal service,
- Scaled service,
- Cell ([WSO2 definition](#)) (service of services) / Domain ([Uber definition](#)) / Cluster [[DEDS](#)].

Examples:

- Service-Based Architecture [[FSA](#) but not DEDS],
- Microservices [[MP](#), [FSA](#)],

- [Actors](#),
- (inexact) Nanoservices (API layer),
- (inexact) Device Drivers.

Structure: A component per subdomain.

Type: Main.

Benefits	Drawbacks
Supports large codebases	Global use cases are hard to debug
Multiple development teams and technologies	Poor latency in global use cases
Forces may vary by subdomain	No good way to share state between services The domain structure should never change Operational complexity

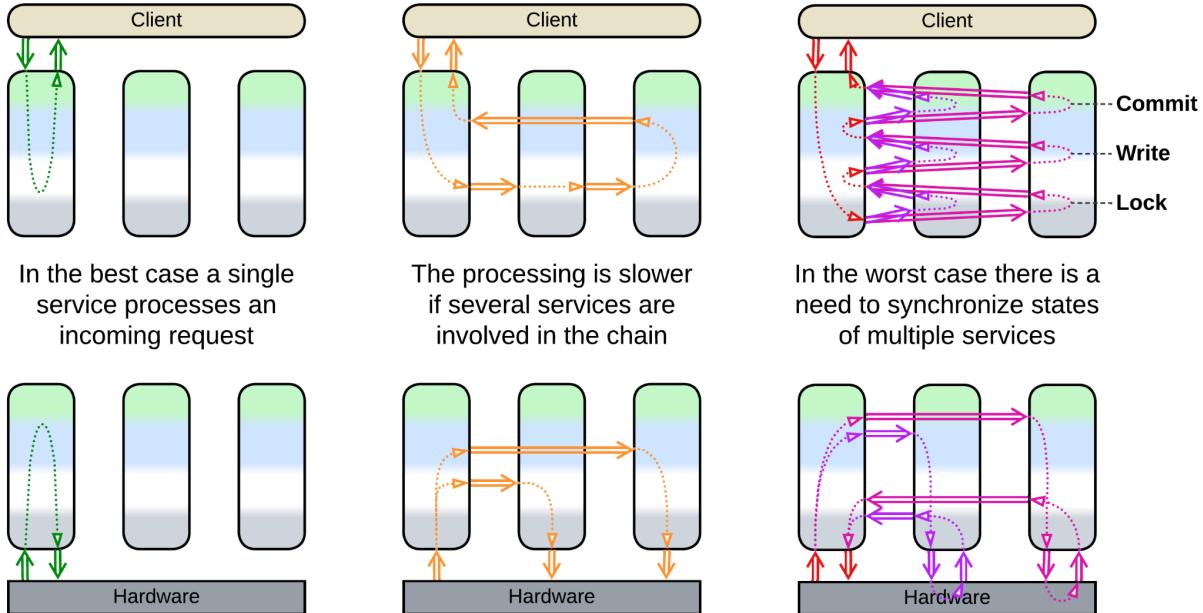
References: [\[FSA\]](#) has a chapter on *Service-Based Architecture*; [\[MP\]](#) is dedicated to *Microservices*.

Splitting a [Monolith](#) by *subdomain* allows for mostly independent properties, development, and deployment of the resulting components. However, for the system to benefit from the division, the subdomains must be loosely coupled and, ideally, of comparable size. In that case the partitioning can reduce complexity of the project's code by cutting accidental dependencies between the subdomains. Moreover, if one of the resulting services grows unmanageably large, it can often be further partitioned by sub-subdomains to form a [Cell](#). This flexibility is paid for through the complexity and performance of use cases which involve multiple subdomains. Another issue to remember is that boundaries between services are [nearly impossible](#) to move at later project stages as the services grow to vary in technologies and implementation styles, thus separation into services assumes perfect practical knowledge of the domain and relatively stable requirements.

Research shows that when more than five programmers work on the same subject, their performance degrades. Therefore, if we want our employees to be efficient, they should be grouped into small teams and each team should be given ownership of a dedicated component.

Performance

Interservice communication is relatively slow and resource-consuming, therefore it should be kept to a minimum.



The perfect case is when a single service has enough authority to answer a client's request or process an event. That case should not be that rare as a service covers a whole subdomain while subdomains are expected to be loosely coupled (by definition).

Worse is when an event starts a chain reaction throughout the system, likely looping back a response to the original service or changing the target state of another controlled subsystem.

In the slowest scenario a service needs to synchronize its state with multiple other services, usually via *locks* and *distributed transactions*.

Multiple [instances](#) of an individual service may be deployed to improve throughput of the system. However, such a case will likely need a [Middleware](#) or [Load Balancer](#) to distribute interservice requests among the instances and a [Shared Repository](#) to store and synchronize any non-shardable (accessed by several instances) state.

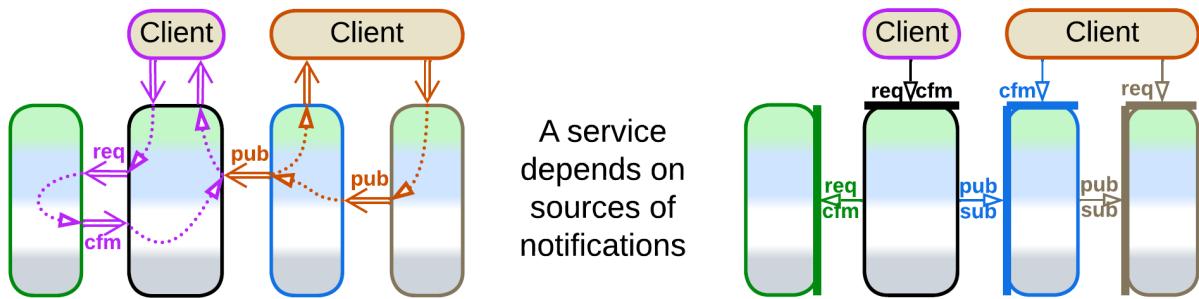
Dependencies

When we see a service to *request* help from other services and then receive the results (in a *confirmation* message), that service [orchestrates](#) the services it uses. Services often orchestrate each other because the subdomain a service is dedicated to is not independent of other subdomains.

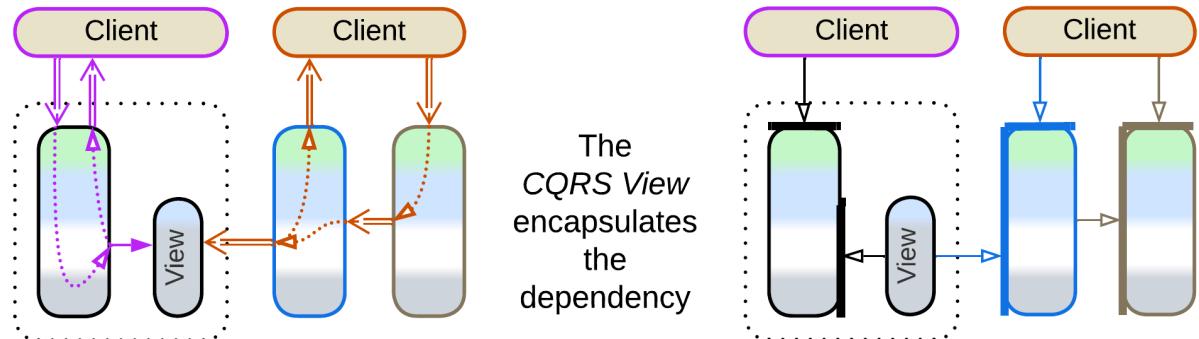


Another way for services to communicate is [choreography](#) – when a service sends a *command* or publishes a *notification* and does not expect any response. This is characteristic of [Pipelines](#) which are covered in the next chapter. Right now we should note

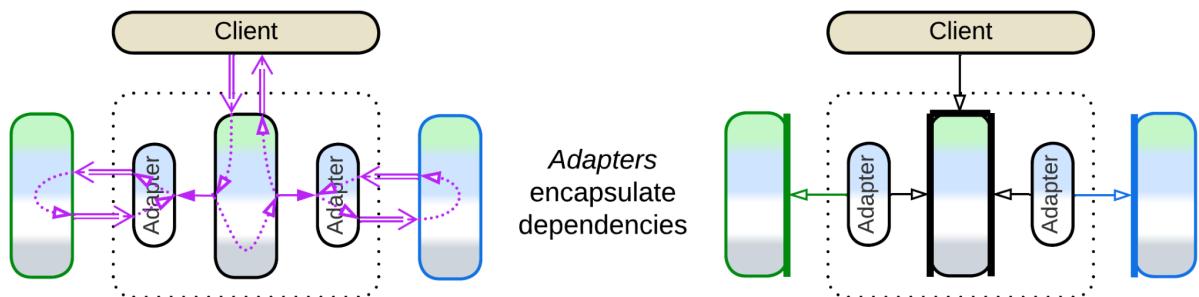
that orchestration and choreography may be intermixed, in which case a service depends on all the services it uses or subscribes to.



If the system relies on notifications (services publish *domain events*), it is possible to avoid interservice *queries* (pairs of a *read* request and confirmation with the data retrieved) by aggregating data from notifications in a [CQRS \(or materialized\) View \[MP\]](#), which can reside [in memory](#) or in a database. Views can be planted inside every service that needs data owned by other services or can be gathered into a dedicated [Query Service \[MP\]](#). Though the main goal of CQRS Views is to resolve distributed joins from databases of multiple services, they also help [remove dependencies](#) in the code of services and optimize out interservice queries, simplifying APIs and improving performance. Further examples will be discussed in the chapter on [Polyglot Persistence](#).



In general, a large service should wrap its dependencies with an [Anticorruption Layer \[DDD\]](#), following the ideas of [Hexagonal Architecture](#). The layer consists of *Adapters* [[GoF](#)] between the internal domain model of the service and the APIs of the components it uses. The *Adapters* isolate the business logic from the external environment, granting that no change in the interface of an external service or library may ever take much work to support on the side of the team that writes our business logic as all the ensuing updates are limited to a small adapter.



Applicability

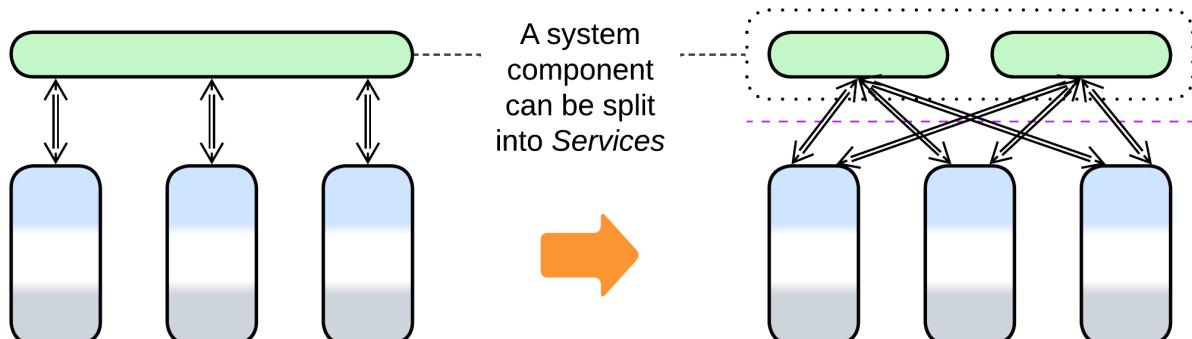
Services are good for:

- *Large projects.* With multiple services developed independently, a project may grow well above 1 000 000 lines of code and still be comfortable to work on as every team needs to know only the medium-sized component it owns.
- *Specialized teams.* Each service would often be written and supported by a dedicated team that invests its time in learning its subdomain. This way no one needs to have a detailed knowledge of the full set of requirements, which is next to impossible in large domains.
- *Varied forces.* In system and embedded programming, components of wildly varying behaviors need to be managed. Each of them is controlled by a dedicated service (called *driver*) which adapts to the specifics of the managed subsystem.
- *Flexible scaling.* Some services may be under more load than others. It makes sense to deploy multiple instances of heavily loaded services.

Services should be avoided in:

- *Cohesive domains.* If everything strongly depends on everything, any attempt to cut the knot with interfaces is going to make things worse unless the project is already dying because of its huge codebase, in which case you have nothing to lose.
- *Unfamiliar domains.* If you don't understand the intricacies of the system you are going to build, you may misalign the interfaces and, by the time that the mistakes come to light, the architecture will be too hard to change [LDDD]. The coupled Services you get may actually be worse than a Monolith.
- *Quick start.* It takes effort to design good interfaces and contracts for Services and managing multiple deployment units is not free of trouble. Debugging will be an issue.
- *Low latency.* If the system as a whole needs to react to events in real time, complex services should be avoided. Nevertheless, an individual service can provide low latency for local use cases (when a single service has enough authority to react to the incoming event), wherefore simple non-blocking actors are widely used in control software.

Relations



- Division by subdomain can be applied to Layers to form Service-Oriented Architecture (layers of services); to Proxy, Orchestrator, or API Gateway to make Backends for Frontends; to a (shared) database resulting in Polyglot Persistence.
- Services can be extended with a Proxy, Orchestrator, Middleware, or Shared Repository.

- Each service can be implemented by [Layers](#) (making a [layered service](#)), [Hexagonal Architecture](#), or a [Cell](#).

Variants by isolation

Division by subdomain is so commonplace and varied that no universal terminology emerged over the years. Below is my summary, in no way complete, of several ways such systems vary. Each section lists the well-known architectures it applies to.

First and foremost, there are multiple grades between a cohesive [Monolith](#) and distributed [Services](#). You should choose incrementally when to stop because the benefits of these next stages (color-coded below) may not outweigh their drawbacks for your project.

I review here only the most common options while a few more esoteric architectures are found in [Volodymyr Pavlyshyn's overview](#).

Synchronous modules: Modular Monolith (Modulith)

The first stage to take when designing a large project is the division of the codebase into loosely coupled modules that match subdomains (*bounded contexts* [[DDD](#)]). If successful, that parallelizes development to a team per module while the entire application still runs in a single process, thus it stays easy to debug, the modules can share data, and any crash kills the whole system (you don't need to take care of partial failures). You pay by establishing boundaries which will not be easy to move in the future.

<i>Benefits</i>	<i>Drawbacks</i>
Multi-team development	Subdomain boundaries are settled

Asynchronous modules: Modular Monolith (Modulith), Embedded Actors

The next stage is separating the modules' execution threads and data. Each module becomes a kind of [actor](#) that communicates with other components through messaging. Now your modules don't block each other's execution and you can [replay events](#) at the cost of nightmarish debugging and no clean way to share data between or synchronize the state of the components.

<i>Benefits</i>	<i>Drawbacks</i>
Multi-team development	Subdomain boundaries are settled
Event replay	No good way to share data or synchronize state
Some independence of module qualities	Hard to debug

Multiple processes

There is also the option of running system components as separate binaries which lets them vary in technologies, allows for granular updates, and addresses stability (a web browser does not stop when one of its tabs crashes). But it adds a whole dimension of error recovery and partially executed scenarios. Moreover, divergency of technologies makes moving pieces of code between the services impossible.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No good way to share data or synchronize state
Independence of component qualities and technologies	Hard to debug
Single-component updates	Needs error recovery routines
Software fault isolation	Data inconsistencies after partial crashes
Limited granular scalability	

Distributed runtime: Function as a Service (FaaS) (including Nanoservices), Backend Actors

Modern distributed [runtimes](#) create a virtual namespace that may be deployed on a single machine or over a network. They may redistribute running components over servers in a way to minimize network communication and may offer distributed debugging. With [Actors](#), if one of them crashes, that generates a message to another actor which may decide on how to handle the error. The convenience of using a runtime has the dark side of vendor lock-in.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No good way to share data or synchronize state
Independence of component qualities and technologies	Hard to debug
Single-component updates	Needs error recovery routines
Full fault isolation	Data inconsistencies after partial crashes
Full dynamic granular scalability	Vendor lock-in Moderate communication overhead Moderate performance overhead caused by the framework

Distributed services: Service-Based Architecture, Space-Based Architecture, Microservices

Fully autonomous services run on dedicated servers or virtual machines. This way you employ resources of multiple servers, but the communication between them is both unstable (requests may be lost, reordered or duplicated) and slow and debugging tends to be very hard. [Mesh](#)-based ([Microservices](#)) and [Space-Based](#)) architectures provide dynamic scaling under load.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No good way to share data or synchronize state
Independence of component qualities and technologies	Very hard to debug
Single-component updates	Needs error recovery routines
Full fault isolation	Data inconsistencies after partial crashes

Full (dynamic for *Mesh*) granular scalability High communication overhead

Variants by communication

Services also differ in the way they communicate which influences some of their properties:

Direct method calls

When components run inside the same process and share execution threads, one component can call another. That is blazingly fast and efficient, but you should take care to protect the module's state from simultaneous access by multiple threads (and yes, [deadlocks](#) do happen in practice). Moreover, it is hard to know what the module you call is going to call in its turn, while you are waiting on it – thus no matter how much you optimize your code, its performance depends on that of other components, often in subtle ways.

RPCs and commands (request/confirm pairs)

If a service [calls into another service](#) or requests it to act and return results (this is how method calls are implemented in distributed systems) it has to store the state of the scenario it is executing for the duration of the call (until the confirmation message is received). That uses resources: the stored state is kept in RAM and the interruption and resumption of the execution wastes CPU cycles on context switch and on the resulting cache misses. Blocked threads are especially heavy while coroutines or fibers are more lightweight but are still not free.

Another trouble with distributed systems comes from error recovery: if your component did not receive a timely response, you don't know if your request was (or is being, or will be) executed by its target – and you need to be really careful about possible data corruption if you retry it and it is executed twice [[MP](#)].

If a request is duplicated (as a slow network, overloaded service, or lost confirmation may cause a retry), it is important to make sure that the second (or parallel) execution of the request does not change the system's data. This is achieved either by using [idempotent](#) logic (which is based on assignment instead of increasing or decreasing values in place), or by writing the id of the last processed message to the database (and checking that the incoming message's id is greater than the one found in the database) [[MP](#)].

On the bright side, [orchestration](#) is human- and debugger-friendly as it keeps consecutive actions close together in the code. Therefore, synchronous interaction is the default mode of communication in many projects.

Notifications (pub/sub) and shared data

A service may do something, publish a notification or write results to a shared datastore for other services to process, and forget about the task as it has completed its role. [Choreography](#) is resource-efficient, but you need to find and read multiple pieces of code which are spread out over several services to understand or debug the whole use case.

(inexact) No communication

Finally, some kinds of services, namely [device drivers](#) and [Nanoservices](#), never communicate with each other. Strictly speaking, such services don't make a system – instead, they are isolated *Monoliths* which are managed by a higher-level component (OS kernel for drivers, client for *Nanoservices*).

Nevertheless, it is a fun fact that if the services don't intercommunicate, the main drawbacks of the Services architecture disappear:

- There is no slow and error-prone interservice communication (they never communicate!).
- It's not hard to debug multi-service use cases (there are no such scenarios!).
- The services don't corrupt data on crash (there are no distributed transactions).

Variants by size

Last but not least, the simplest classification of subdomain-separated components is by their size:

Whole subdomain: (Sub-)Domain Services

Each *Domain Service* [[FSA](#)] of [Service-Based Architecture](#) [[FSA](#)] implements a whole subdomain. It is the product of the full-time work of a dedicated team. A project is unlikely to have more than 10 of such services (in part because the number of top-level subdomains in any domain is usually limited).

Part of a subdomain: Microservices

Microservices enthusiasts estimate the best size of a component of their architecture to be below a month of development by a single team. That allows for a complete rewrite instead of refactoring in case the requirements change. When a team completes one microservice it can start working on another, probably related, one while still maintaining its previous work. A project is likely to contain from tens to few hundreds of microservices.

Class-like: Actors

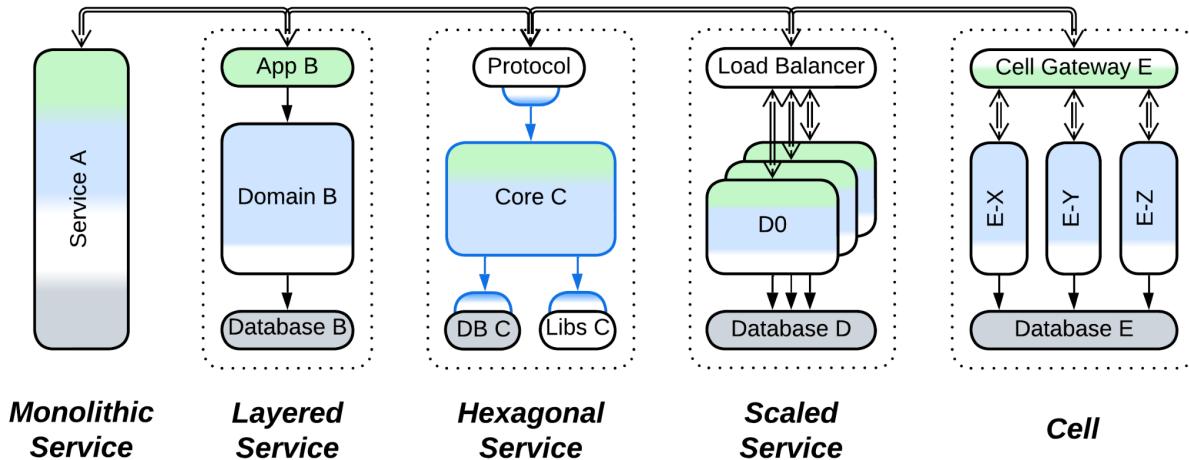
An [actor](#) is an object with a message-based interface. They are used correspondingly. Though the size of an actor may vary, as does the size of an OOP class, it is still very likely to be written by a single programmer.

Single function: FaaS, Nanoservices

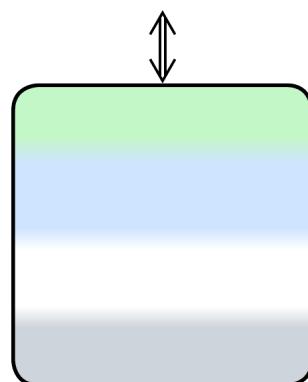
A [nanoservice](#) is a single function ([FaaS](#) [[DDS](#)]) usually deployed to a serverless provider. Nanoservices are used as API method handlers or as building blocks for [Pipelines](#).

Variants by internal structure

A service is not necessarily monolithic inside. Because a service is encapsulated from its users by its interface, it can have any kind of internal structure. The most common cases, which can be intermixed together, are:

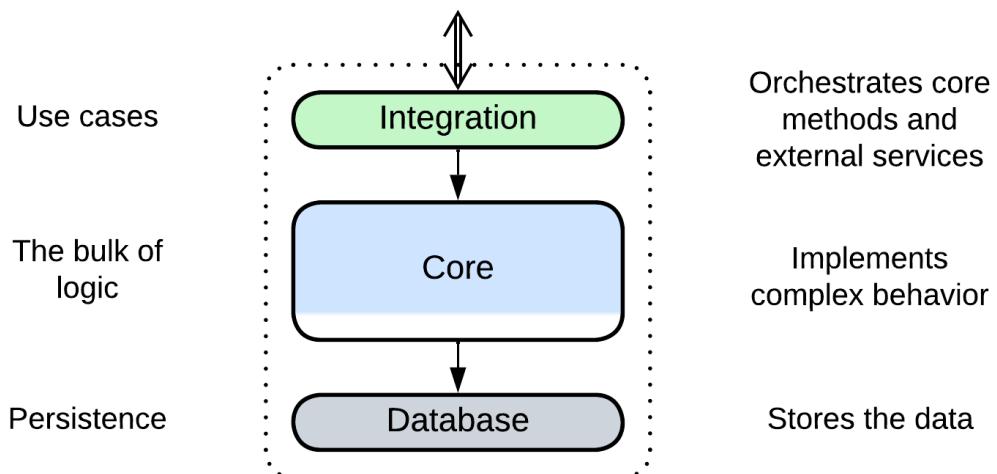


Monolithic service



A *monolithic service* is a service with no definite internal structure, probably small enough to allow for complete rewrite instead of refactoring – the ideal of proponents of [Microservices](#). It is simple & stupid to implement but relies on external sources of persistent data. For example, *device drivers* and [Actors](#) usually get their (persisted) configuration during initialization. A monolithic backend service may receive all the data it needs in incoming requests, via a query to another service, or by reading it from a [Shared Database](#).

Layered service

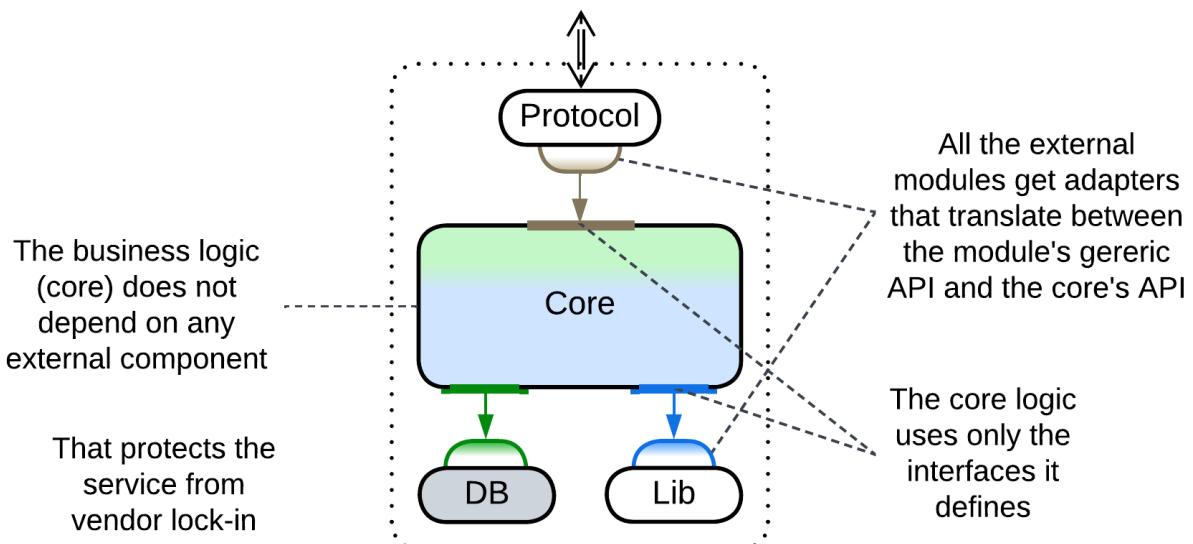


A *layered service* is [divided into layers](#). This approach is very common both with backend (*micro*-)services, where at least the database is separated from the business logic, and with *device drivers* in system programming, where hardware-specific low-level interrupt handlers and register access are separated from the main logic and high-level OS interface.

Layering provides all of the benefits from the [Layers](#) pattern, including support for [conflicting forces](#), which may manifest, for example, as the ability to deploy the database to a dedicated server in backend or as a very low latency in the hardware-facing layer of a device driver.

Another benefit comes from the existence of the upper integration layer which may [orchestrate interactions with other services](#), isolating the lower layers from external dependencies.

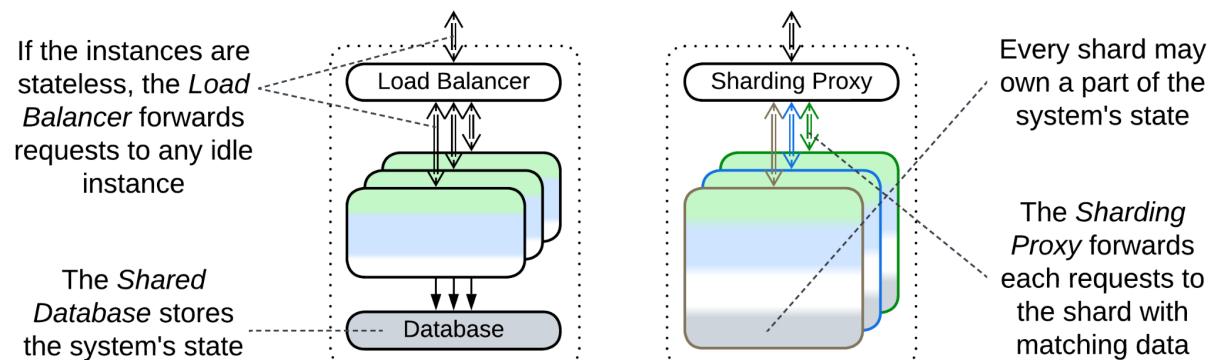
Hexagonal service



A *hexagonal service* has its external dependencies isolated behind vendor-agnostic interfaces.

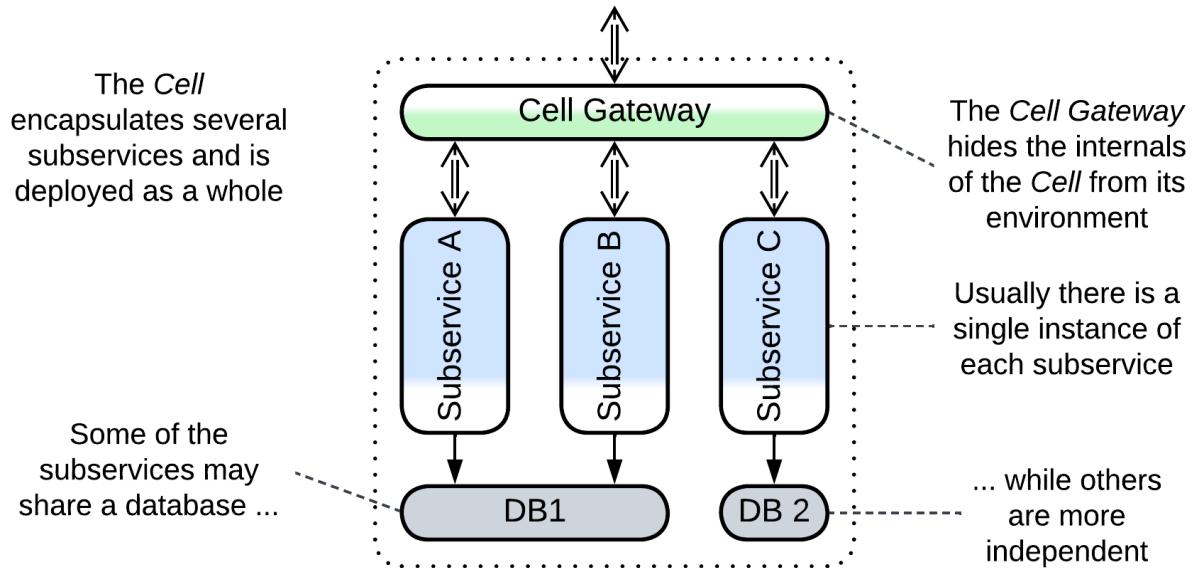
This is a real-world application of [Hexagonal Architecture](#) which both ensures that the business logic does not depend on specific technologies and protects from vendor lock-in. It is highly recommended for long-lived projects.

Scaled service



With scaled services there are multiple [instances](#) of a service. In most cases they [share a database](#) (though sometimes the database may be [sharded](#) or [replicated](#) together with the service that uses it) and get their requests through a [Load Balancer](#) or [Sharding Proxy](#).

Cell (WSO2 definition) (service of services), Domain (Uber definition), Cluster



When a service is split into a set of subservices, it makes a [Cell](#) (WSO2 name), [Domain](#) (Uber name), or [Cluster](#) [[DEDS](#)]. All the incoming communication passes through a [Cell Gateway](#) which encapsulates the *Cell* from its environment. Outgoing communication may involve the *Cell Gateway* or dedicated [Adapters](#) ([Anticorruption Layer](#) [[DDD](#)]). A *Cell* may deploy its own [Middleware](#) and/or [share a database](#) among its components.

[Cell-Based Architecture](#) (according to WSO2), as opposed to [Amazon's alias](#) for [Shards](#)) appears when there is a need to recursively split a service, either because it grew too large or because it makes sense to use several incompatible technologies for its parts. It may also be applied to group services if there are too many of them in the system.

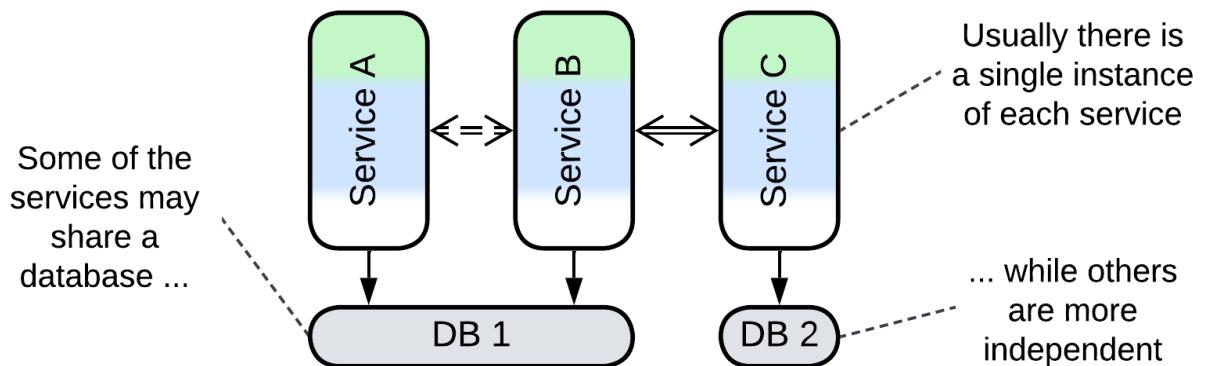
[Domain-Oriented Microservice Architecture](#) (DOMA) is a [SOA](#)-style layered system of *Cells*.

Examples

Services are pervasive among advanced architectures which either build around a layer of services that contains the bulk of the business logic ([Proxy](#), [Orchestrator](#), [Middleware](#) and [Shared Repository](#)) or use small services as an extension of the main monolithic component ([Plugins](#) and [Hexagonal Architecture](#)). [Polyglot Persistence](#), [Backends for Frontends](#) and [Service-Oriented Architecture](#) go all out partitioning the system into interconnected layers of services. [Hierarchy](#) and [Mesh](#) require the services to implement or use a polymorphic interface to simplify the components that manage them.

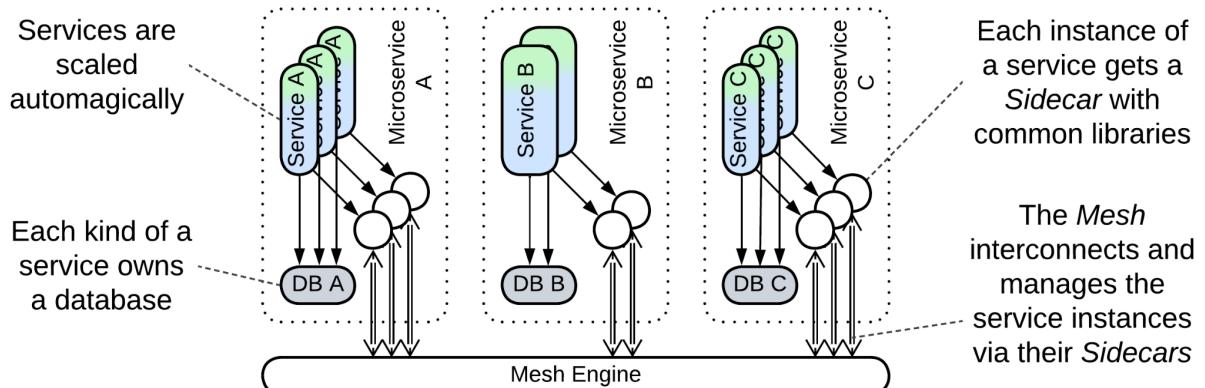
Examples of Services include:

Service-Based Architecture



This is the simplest use of Services where each subdomain gets a dedicated component. A *Service-Based Architecture* [FSA] tends to consist of a few coarse-grained services, some of which may [share a database](#) and have little direct communication. An [API Gateway](#) is often present as well.

Microservices



Microservices [MP, FSA] are usually smaller than components of *Service-Based Architecture* and feature multiple services per subdomain with strict decoupling: no [Shared Database](#), independent (and often dynamic) scaling and deployment. Even [orchestration](#) and distributed transactions ([Sagas](#)) are considered to be a smell of bad design.

Microservices fit loosely coupled domains with parts which [vary drastically](#) in both forces and technologies. Any attempt to use them for an unfamiliar domain is [calling for trouble](#). Some authors insist that the “micro-” means that a microservice should not be larger in scope than a couple of weeks of work for a programming team. That allows rewriting one from scratch instead of refactoring. Others assert that too high a granularity makes everything [overcomplicated](#). Such a diversity of opinions may mean that the applicability and the very definition of *Microservices* varies from domain to domain.

This architecture usually relies on a [Service Mesh](#) for [Middleware](#) where common functionality, like logging, is implemented in co-located [Sidecars](#) [DDS]. A layer of [Orchestrators](#) (called *Integration Microservices*) [may be present](#), resulting in [Cell-Based Architecture](#) or [Backends for Frontends](#).

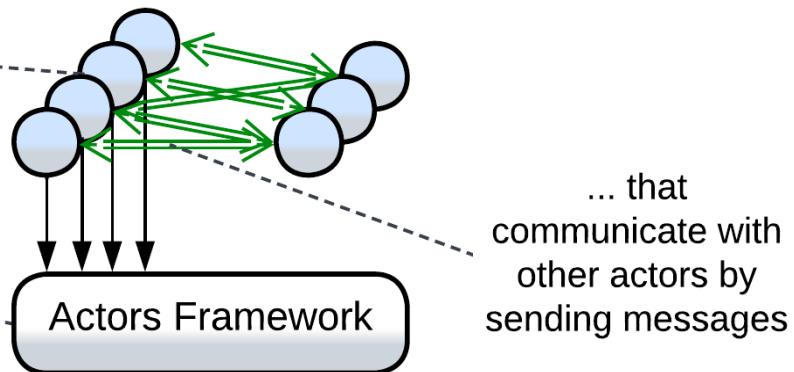
Dynamically scaled [Pools](#) of service instances are common thanks to the elasticity of hosting in a cloud. Extreme elasticity requires [Space-Based Architecture](#), which puts a [distributed in-memory database](#) node in each *Sidecar*.

Some authors [distinguish](#) between *architectural patterns* and *architecture styles (architectures)* [[FSA](#), [MP](#)]. The difference is similar to that between libraries and frameworks: you use a library or pattern (e.g. division of a component into [Layers](#) or [Services](#)) when you think that it will help your needs, but you build your entire system according to the rules of a framework or style (such as [Microservices](#) or [Enterprise SOA](#)). This book does not accent that difference – instead, it boils down styles to combinations of patterns.

Actors

There are often many instances of an actor ...

The underlying framework may feature a *Shared Database*



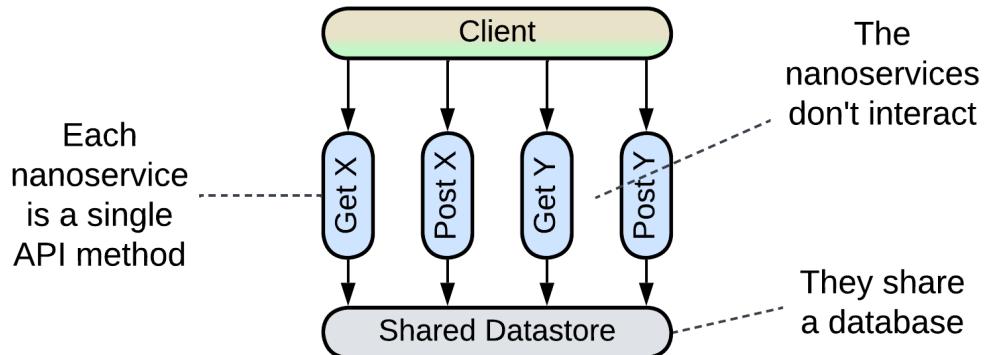
An [actor](#) is an entity with private data and a public message queue. They are like objects with the difference that actors communicate only by sending each other asynchronous messages. The fact that a single execution thread may serve thousands of actors makes actor systems an extremely lightweight approach to asynchronous programming. As an actor is usually single-threaded, there is no place for *mutexes* and *deadlocks* in the code and it is possible to [replay events](#). Non-blocking [Proactors](#) are often used in [real-time systems](#).

Actors have long been used in telephony (which is the domain where real-time communication meets complex logic and low resources) and with the invention of distributed runtime environments (e.g. Erlang/OTP or Akka) they found their place in messengers and banking which need to interconnect millions of users while providing personalized experience and history for everyone. Every user gets an actor that represents them in the system by communicating both with other actors (forming a kind of [Mesh](#)) and with the user's client application(s).

If we apply a bit of generalization, we can deduce that any server or backend service is an actor because its data cannot be accessed from outside and asynchronous IP packets are its only means of communication. Services of [Event-Driven Architecture](#) closely match this definition.

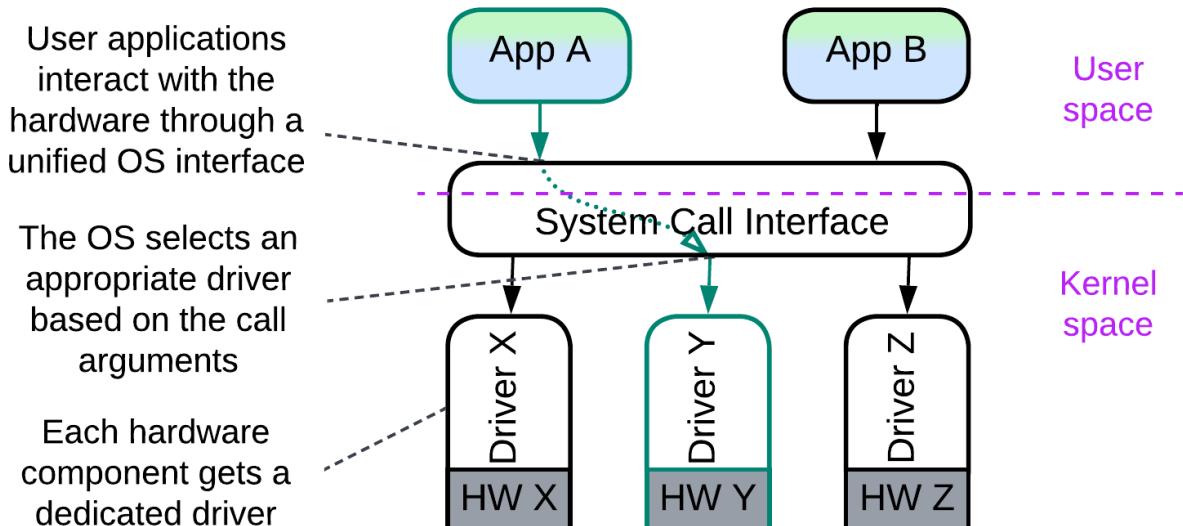
A [deadlock](#) happens when several threads in a system wait for each other to release unique resources they have each taken. As no thread involved in the [deadlock](#) can continue its operation, the system cannot complete its task. A single-threaded actor cannot [deadlock](#) because it does not contain multiple threads in the first place.

(inexact) Nanoservices (API layer)



Though *Nanoservices* are defined by their size (a single function), not system topology, I want to mention a specific application from Diego Zanon's book *Building Serverless Web Applications*. That example is interesting because it comprises a single layer of isolated functions (each providing a single API method) which may share functionality by including code from a common repository. As nanoservices of this kind never interact, the common drawbacks of Services (poor debugging and high latency) don't apply to them.

(inexact) Device Drivers



An operating system must run efficiently with an unpredictable combination of hardware components, any of which can come from different manufacturers. It is impossible to know all the combinations beforehand. Thus it employs one service (called *driver*) per hardware device. A driver *adapts* a manufacturer- and model-specific hardware interface to the generic interface of the OS *kernel*, allowing for the kernel to operate the hardware it controls without the detailed knowledge of the model. Internally, a driver is usually *layered*:

- The lowest layer, called the [Hardware Abstraction Layer](#) (HAL), provides a model-independent interface for a whole family of devices from a manufacturer.
- The next layer of a driver is likely to contain manufacturer-specific algorithms for efficient use of the hardware.
- The third layer, if present, is probably busy with high-level tasks which are common for all devices of the given type and may be implemented by the kernel programmers.

The whole system of kernel, drivers, and user applications comprises the [Microkernel](#) architecture which bridges resource consumers and resource providers. As the drivers don't

need to coordinate themselves (this is done by the kernel), they don't really make a system of Services and thus don't have the corresponding drawbacks.

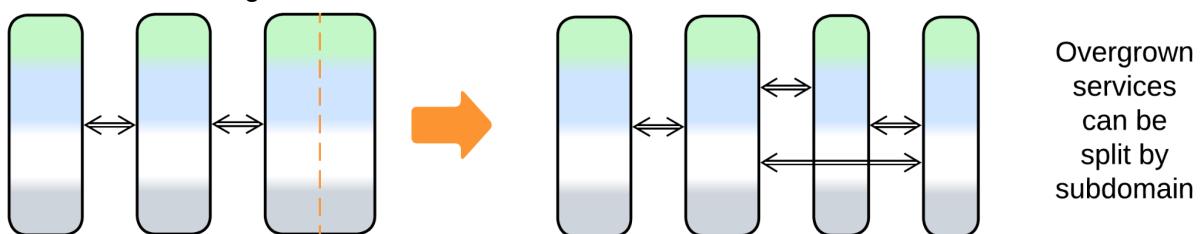
Evolutions

Services are subject to a wide array of evolutions, just like the other basic metapatterns. These are summarized below and detailed in [Appendix E](#).

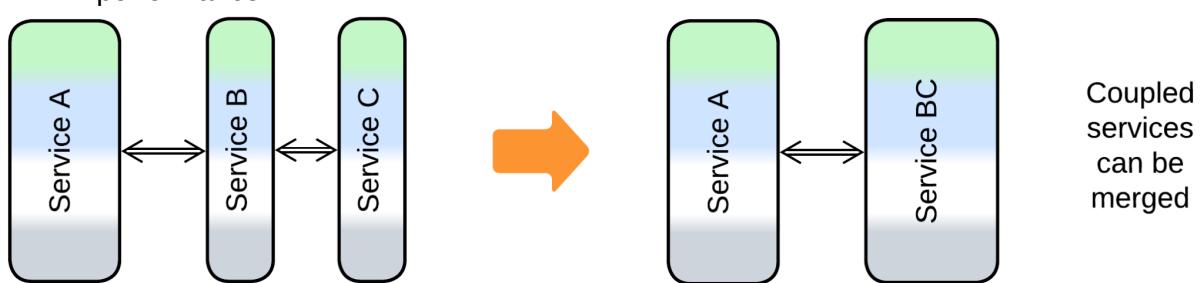
Evolutions that add or remove services

Services work well when each service matches a subdomain and is developed by a single team. If those premises change, you'll need to restructure the services:

- A new feature request may emerge outside of any of the existing subdomains, creating a new service, or a service may grow too large to be developed by a single team, calling for division.



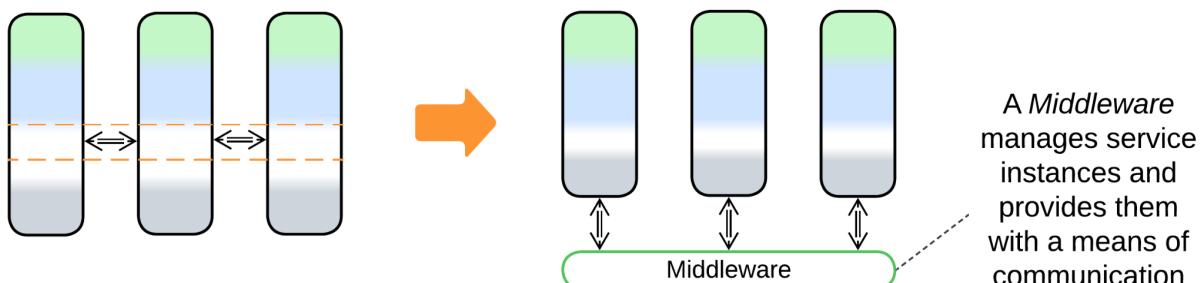
- Two services may become so strongly coupled that they fare better if merged together, or the entire system may need to be glued back into a [Monolith](#) if the domain knowledge changes or if interservice communication strongly degrades performance.



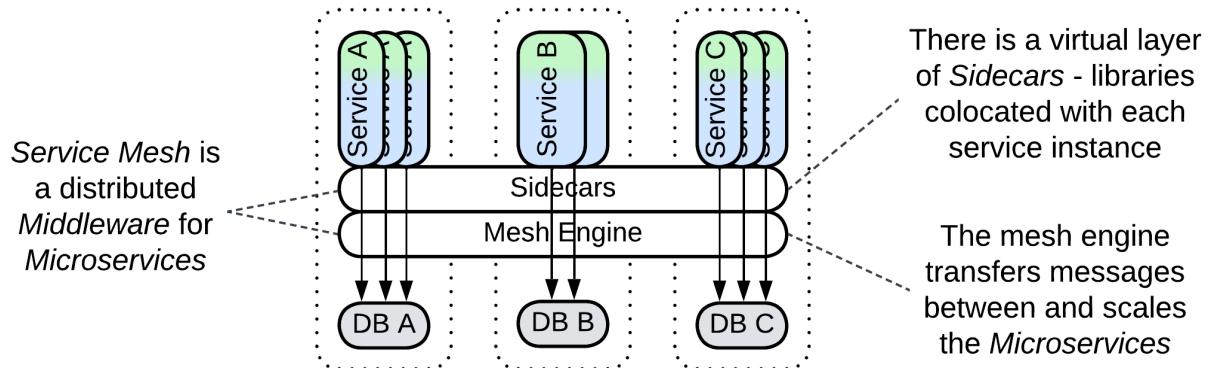
Evolutions that add layers

The most common modifications of a system of Services involve supplementary system-wide layers which compensate for the inability of the services to [share](#) anything among themselves:

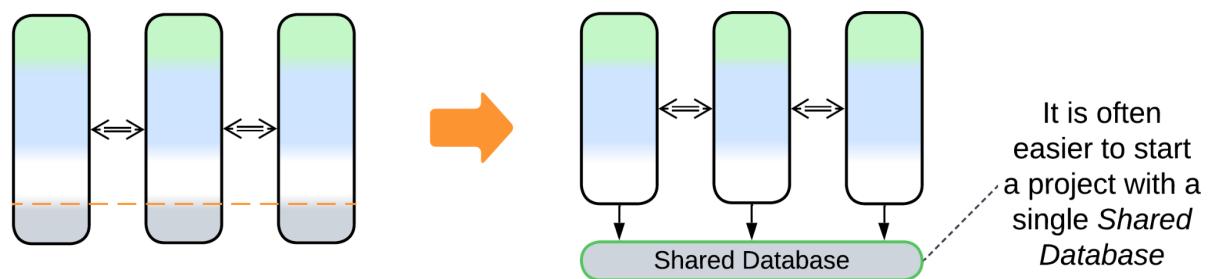
- A [Middleware](#) tracks all the deployed service instances. It mediates the communication between them and may manage their scaling and failure recovery.



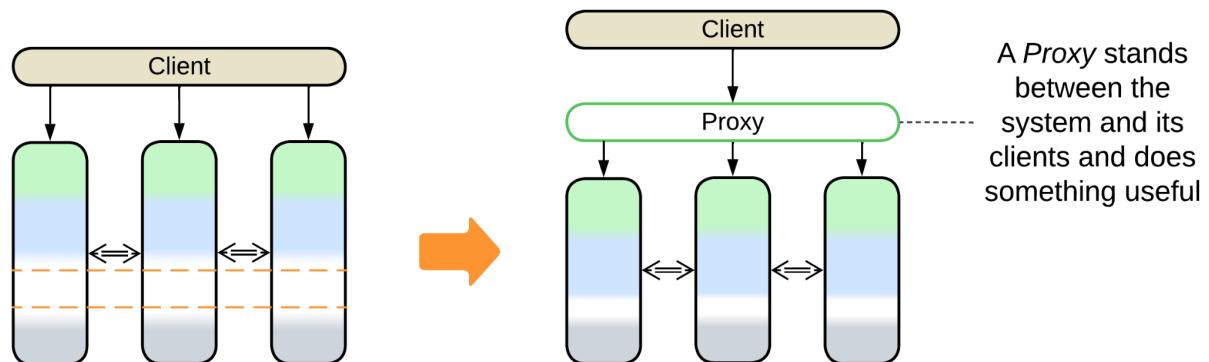
- [Sidecars \[DDS\]](#) of a [Service Mesh](#) make a virtual layer of shared libraries for the [Microservices](#) it hosts.



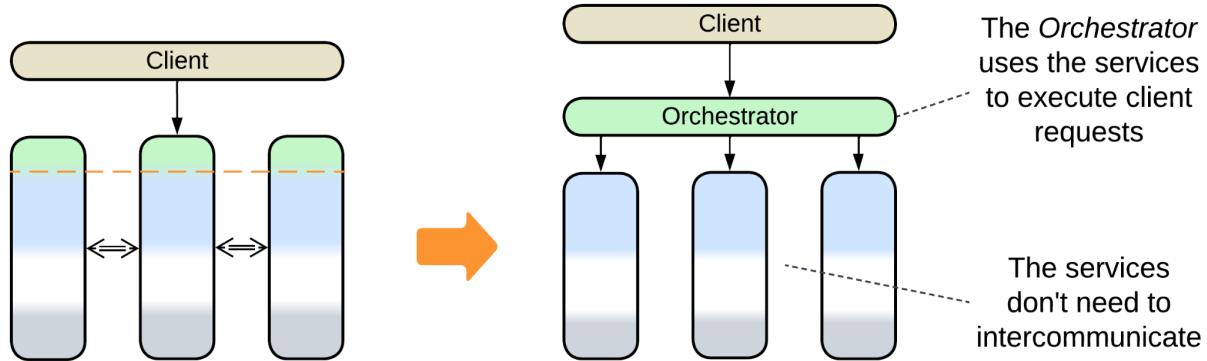
- A [Shared Database](#) simplifies the initial phases of development and interservice communication and enables the use of Services in [data-centric domains](#).



- [Proxies](#) stand between the system and its clients and take care of shared aspects that otherwise would need to be implemented by every service.



- An [Orchestrator](#) is the single place where the high-level logic of all use cases resides.



Those layers may also be consolidated into [Combined Components](#):

- [Message Bus](#) is a type of [Middleware](#) that supports multiple protocols.
- [API Gateway](#) combines [Gateway](#) (a kind of [Proxy](#)) and [Orchestrator](#).
- [Event Mediator](#) is an [orchestrating Middleware](#).
- [Shared Event Store](#) combines [Middleware](#) and [Shared Repository](#).
- [Enterprise Service Bus \(ESB\)](#) is an [orchestrating Message Bus](#).
- [Space-Based Architecture](#) employs all four layers: [Gateway](#), [Orchestrator](#), [Shared Repository](#), and [Middleware](#).

Evolutions of individual services

Each service starts as either a [Monolith](#) or as [Layers](#) and may undergo the corresponding evolutions:

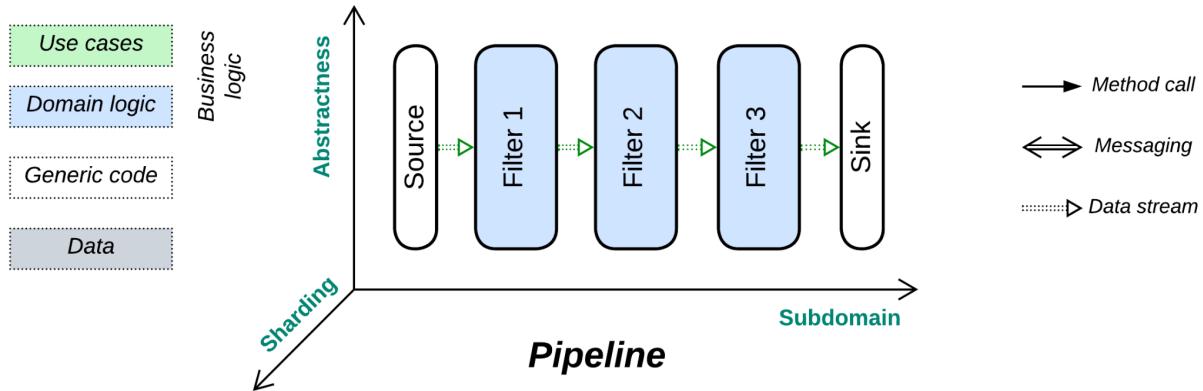
- [Layers](#) help to reuse third-party components (e.g. a database), organize the code, support conflicting forces and the upper layer of the service may [orchestrate other services](#).
- A [Cell](#) is a service which is subdivided into several services that share an [API Gateway](#) and may [share a database](#) and/or a [Middleware](#). All of the components of a [Cell](#) are usually deployed together. That helps when dealing with overgrown services without increasing the operational complexity of the system – but only if the [Cell's](#) components are loosely coupled.
- A service may use a [Load Balancer](#) or a load balancing [Middleware](#) to scale. Its [instances](#) usually rely on a [Shared Database](#) for persistence.
- [Polyglot Persistence](#) or [CQRS](#) may be used inside a service to improve the performance of its data layer.
- [CQRS Views \[MP\]](#) or a [Query Service \[MP\]](#) help reconstruct the state of other services from event sourcing.
- [Hexagonal Architecture](#) isolates the business logic of the service from external dependencies.
- In rare cases [Plugins](#) or [Scripts](#) help to vary the behavior of a service.

Summary

Services deal with large projects by dividing them into subdomain-aligned components of smaller sizes which can be handled by dedicated teams. These may vary in technologies and quality attributes. However, services have a hard time cooperating in anything, from sharing data to debugging, and come with an innate performance penalty. There are a few

options halfway between [*Monolith*](#) and *Distributed Services* that have milder benefits and drawbacks.

Pipeline



Never return. Push your data through a chain of processors.

Known as: Pipeline [[DDS](#)].

Variants:

By scheduling:

- Stream processing [[DDIA](#)] / Nearline system [[DDIA](#)],
- Batch processing [[DDIA](#), [DDS](#)] / Offline system [[DDIA](#)].

Examples:

- Pipes and Filters [[POSA1](#), [POSA4](#), [EIP](#)] / Workflow System [[DDS](#), [DDIA](#)],
- Choreographed (Broker Topology) Event-Driven Architecture (EDA) [[SAP](#), [FSA](#), [DDS](#)] / [Event Collaboration](#) [[DEDS](#)],
- [Data Mesh](#) [[LDDD](#), [SAHP](#)],
- [Function as a Service](#) (FaaS) [[DDS](#)] / [Nanoservices](#) (pipelined).

Structure: A component per step of data processing.

Type: Main.

Benefits	Drawbacks
It is very easy to add or replace components	Becomes too complex when the number of scenarios grows
Multiple development teams and technologies	Poor latency
Good scalability	Significant communication overhead
The components can be reused	Error handling may be non-trivial
The components can be tested in isolation	

References: *Pipes and Filters* are defined in [[POSA1](#)] and are the foundation for part 3 (Derived Data) of [[DDIA](#)]. [[DDS](#)] has an overview of all kinds of *Pipelines* in general while [[FSA](#)] has a chapter on *Event-Driven Architecture*. [[DEDS](#)] is dedicated to *Event-Driven Architecture*. The [[SAHP](#)] chapter on *Data Mesh* was written by the pattern's author. [[EIP](#)] is a whole book about distributed *Pipelines*.

Pipeline is a variation of [Services](#) with no user sessions [[DDS](#)], a unidirectional data flow, and often a single message type per communication channel (which thus becomes a *data stream*). As processed data does not return to the module that requested processing, there is no common concept of request ownership or high-level (integration, application) business logic, which is instead defined by the graph of connections between the components. On the

one hand, as all the services involved are equal and know nothing about each other (their interfaces are often limited to a single entry point), it is very easy to reshape the overall algorithm. On the other hand, the system lacks the abstractness dimension, thus any new use case builds a separate pipeline which may easily turn the architecture into a mess of thousands of intrinsically interrelated pieces when the number of scenarios grows. Moreover, error handling requires dedicated pipelines that roll back changes to the system's state which had been committed by earlier steps of a failed use case.

Performance

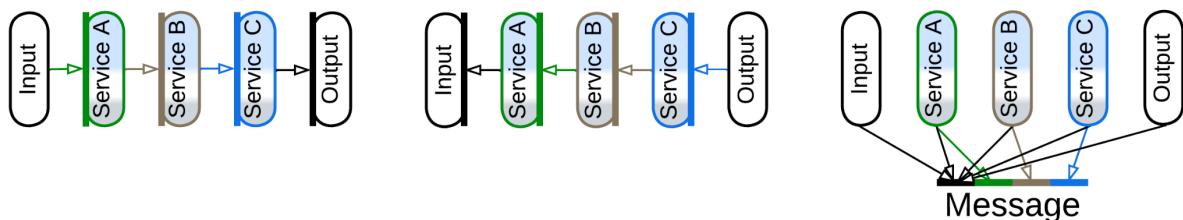
Because any task for a pipeline is likely to involve all (or most of, if branched) its steps, there is no way to optimize away communication. Therefore, latency tends to be high. However, as pipeline components are often stateless, multiple instances of individual services or entire pipelines can run in parallel, making *Pipeline* a highly scalable architecture.

Another point to observe is that a local pipeline naturally spreads the load among the available CPU cores (by using one thread per component) without any explicit locks or thread synchronization.

Dependencies

There are three ways to build communication in a pipeline, each with different dependencies:

- *Commands* make each service depend on the services it sends messages to. It is easy to add a new input to such a pipeline.
- With *publish/subscribe* each service depends on the services it subscribes to. That case favors downstream branching with multiple consumers.
- Services may share a *message schema*, in which case all of them depend on it, not on each other. That allows for reshuffling the services.



Dependencies with Commands

Dependencies with Notifications

Dependencies with Shared Schema

See the [Choreography chapter](#) for more detailed discussion.

Applicability

Pipeline is good for:

- *Experimental algorithms*. This architecture allows for the data processing steps both to be tested in isolation and connected into complex systems without changing the existing code.
- *Easy scaling*. Pipelines tend to evenly saturate all the available CPU cores without any need for custom schedulers. Stateless services can run distributed, thus the *Pipeline*'s scalability is limited only by its data channels.

- *Tailoring projects.* Many pipeline components are abstract enough to be easily reused, greatly reducing the cost of serial development of customized projects once the company builds a collection of common reusable services.

Pipeline does not work for:

- *High number of use cases.* The number of components and their interactions is going to be roughly proportional to the number of supported use cases and will easily overwhelm any developer or architect if new scenarios are added over time.
- *Complex use cases.* Any conditional logic written as two or three lines of code with [*orchestration*](#) is likely to need a separate pipeline and dedicated services with [*choreography*](#). Errors and corner cases are remarkably difficult to handle [[FSA](#)].
- *Low latency.* Every step of a data packet along its journey between services takes time, not in the least because of data serialization. Moreover, the next service in the chain may still be busy processing previous data packets or its activation involves the OS scheduler.

Relations

Pipeline:

- Is a kind of [*Services*](#) with unidirectional communication and often a single input method.
- Is [*involved*](#) in [*CQRS*](#), [*Polyglot Persistence with derived databases*](#), and [*MVC*](#).
- Can be extended with a [*Proxy*](#), [*Middleware*](#), or [*Shared Repository*](#).

Variants by scheduling

A pipeline may be either always active or just run once in a while:

Stream processing, Nearline system

Stream processing [[DDIA](#)] is when the pipeline components (*jobs*, *processors*, *filters*, *services* – whatever you prefer to call them) are actively waiting for input and process each incoming item immediately. This results in *near-real-time* (*nearline*) latency [[DDIA](#)] but the pipeline will then always use resources, even when there is nothing for it to process.

Batch processing, Offline system

Batch processing [[DDIA](#), [DDS](#)] happens when a batch of input items is first collected into storage, and then the pipeline is run (often on a schedule) to process it. Such a mode of action is called *offline* [[DDIA](#)] as the processing results are delayed. However, the company saves on resources because the pipeline is only active for brief periods of time.

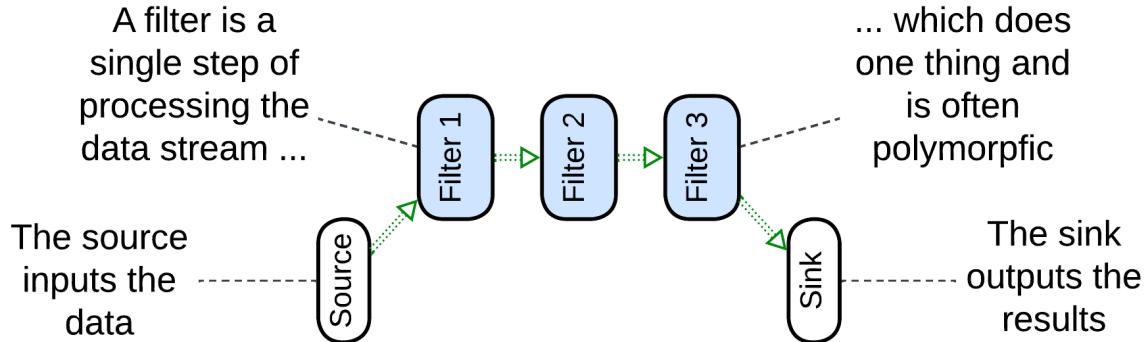
Examples

Pipelines can be local or distributed, linear or branched (usually trees, but cycles may happen in practice), they may utilize a feedback engine to keep the throughput of all their components uniform by slowing down faster steps or scaling out slower ones. In some systems *pipes* (channels) or *filters* (services) persist data. *Pipes* may store the processed data in files or databases to enable error recovery and [*event sourcing*](#). Filters may need to

read or write to a database, which is often *shared*, if the data processing relies on the system's state. Moreover, transferring data through a pipe may be implemented as anything ranging from a method call on the next filter to a pub/sub framework.

Such a variety of options enables the use of pipelines in a wide range of domains. Notwithstanding, there are a few mainstream types of *Pipeline* architectures:

Pipes and Filters, Workflow System



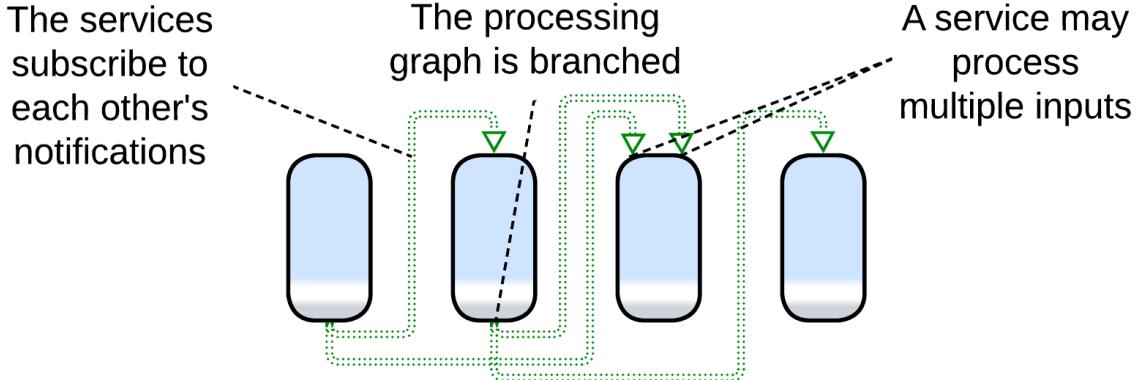
Pipes and Filters [[POSA1](#), [POSA4](#), [EIP](#)] usually name a linear local system which obtains data with its *source*, passes the data through a chain of *filters*, connected by *pipes*, and outputs it via a *sink*. The entire *pipeline* may run as a single process to avoid the overhead of data serialization. It may range from a Unix shell script which passes file contents through a series of utilities to a hardware pipeline for image processing in a video [camera](#). The filters tend to be single-purpose (handle one type of payload) and stateless. In some cases a filter may use dedicated hardware (for encryption or audio/video processing). The entire pipeline often operates a single data format ([Stamp Coupling](#) [[SAHP](#)]).

Though most commonly a filter waits for data to appear in its input pipe, processes it, and pushes the result to its output pipe, thus allowing for multiple filters to run in parallel, some implementations may let the source push the data through the entire pipeline all the way to the sink, with each filter directly calling the next filter in the line or, alternatively, the sink can pull the data by making direct upstream calls [[POSA1](#)]. The last two approaches remove the need for pipes but are then limited to using a single CPU core.

Workflow [[DDIA](#), [DDS](#)] is a more modern name for similar stepwise processing which often stores intermediate results in a file or database and may run distributed. However, the same word generally describes the sequence of high-level steps in a use case [[PEAA](#)], and is another name for *application* or *integration* logic.

Examples: Unix shell pipes, processing of video streams, many types of hardware.

Choreographed (Broker Topology) Event-Driven Architecture (EDA), Event Collaboration



Event-Driven Architecture (EDA) means that the system is built of services which use events to communicate in a non-blocking way. The idea is similar to the [actor model](#) of telecom and embedded programming. Thus, *EDA* itself does not define anything about the structure of the system (except that it is not [monolithic](#)).

In practice, there are [two kinds](#) of Event-Driven Architectures:

- [Choreographed](#) / Broker Topology / [Event Collaboration](#) [[DEDS](#)] – the events are notifications (usually via publish/subscribe) and the services form tree-like structures, matching our definition of *Pipeline*.
- [Orchestrated](#) / Mediator Topology / [Request-Response Collaboration](#) – the events are request/confirmation pairs and usually there is a single entity that drives a use case by sending requests and receiving confirmations. Such a system corresponds to our [Services](#) metapattern with the supervisor being an [Orchestrator](#), discussed in a separate chapter.

An ordinary *Choreographed Event-Driven Architecture* [[SAP](#), [FSA](#), [DDS](#)] is built as a set of subdomain services (similar to those of the parent [Services](#) metapattern). Each of the services subscribes to notifications from other services which it uses as action/data inputs and produces notifications that other services may rely on. For example, an email service may subscribe to error notifications from other services in the system to let the users know about troubles that occur while processing their orders. It will also subscribe to the user data service's add/edit/delete notifications to keep its user contact database updated.

This example shows several differences from a typical *Pipes and Filters* implementation:

- The system supports multiple use cases (e.g. user registration and order processing).
- A service has several entry points (the email service involves an order error handler and user created handler).
- A notification that a service produces may have many subscribers or no subscribers (nobody needs to act on our sending an email to a user).

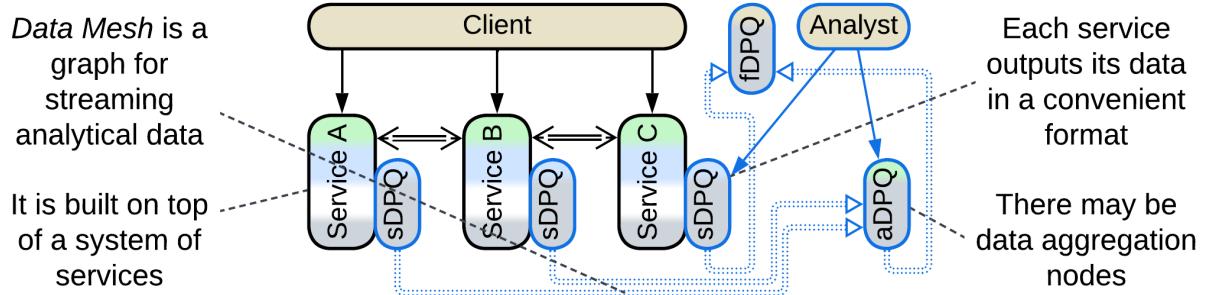
Those points translate to difference in structure: while *Pipes and Filters* is usually a linear chain of components, *EDA* entails multiple branched (and sometimes looped) event flow graphs over a single set of subdomain services.

Pipelined *Event-Driven Architecture* (often boosted with [event sourcing](#)) works well for highly loaded systems of moderate size, but larger projects are likely to grow prohibitively complex graphs of event flows and service dependencies. The architecture's scalability is limited by the services' databases and the pub/sub framework employed.

Event-Driven Architecture may involve a [Gateway](#) as a user-facing event source and sink and a [Middleware](#) for an application-wise pub/sub engine. [Front Controller \[SAHP\]](#) or [Stamp Coupling \[SAHP\]](#) are used if it is important to know the state of requests that are being processed by the pipeline.

Examples: high performance web services.

Data Mesh



First and foremost, [Data Mesh](#) [[LDDD](#), [SAHP](#)] is not a [Mesh](#), but rather a [Pipeline](#). This architecture applies [CQRS](#) on the system level: it separates the interfaces and channels through which the services change their state (matching *commands* or [OLTP](#) of CQRS) and the ones used to retrieve their data (similar to *queries* or [OLAP](#)). That results in two overlapping subsystems, *operational* and *analytical*, that share most of their nodes.

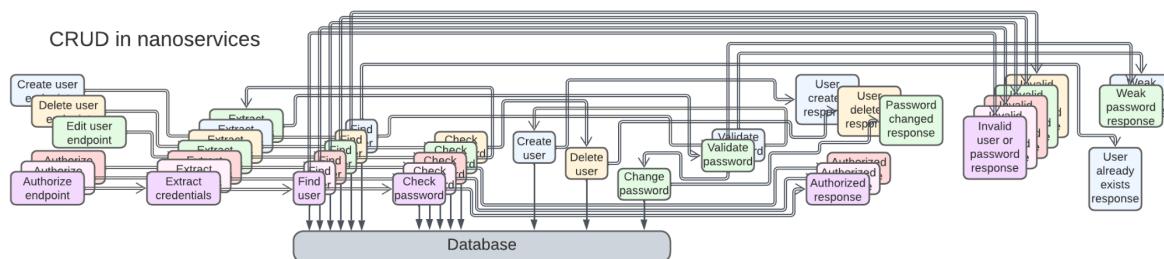
The *operational system* is an ordinary [Microservices](#) or [Event-Driven Architecture](#).

The *analytical system* contains *Data Product Quanta* (*DPQ*) – services that provide convenient access (streaming, replaying, and possibly querying) to parts of the system's data. The *DPQs* are assembled into a graph akin to *Event-Driven Architecture*. There are three kinds of *DPQs* [[SAHP](#)]:

- A *source-aligned (native)* *DPQ* (*sDPQ*) is coupled to an operational service and streams (or provides queries into) its data. It is likely to be implemented as a [Reporting Database](#).
- An *aggregate DPQ* (*aDPQ*) merges and transforms inputs from several sources (*sDPQs* or other *aDPQs*). It is similar to a [Query Service](#).
- A *fit-for-purpose (custom-made)* *DPQ* (*fDPQ*) is an end-user (leaf application) of the *Data Mesh*'s data. It may collect a dataset for machine learning or let a business analyst do their research. *fDPQs* tend to be short-lived one-off components.

There is a pragmatic option to allow an operational service to resort to the analytical system's *DPQs* to query other services' data instead of messaging them directly or implementing a [CQRS View](#) and subscribing it to events that flow in the operational system.

Function as a Service (FaaS), Nanoservices (pipelined)



A [nanoservice](#) is, literally, a [function as a service](#) (FaaS) [DDS] – a stateless (thus perfectly scalable) component with a single input. They can run in proprietary cloud [Middleware](#) over a [Shared Database](#) and are [chained into pipelines](#), one per use case. The code complexity stays low, but as the project grows, the integration will quickly turn into a nightmare of hundreds or thousands of interconnected services.

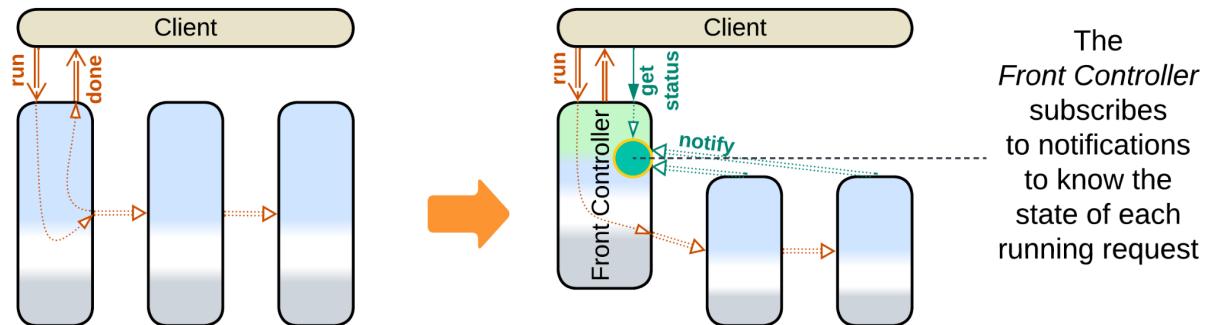
Nanoservices are good for rapid development of small elastic (dynamically scalable) applications. The supported load is limited by the *Shared Database*, and the project evolvability is limited by the complexity of scenarios. As any use case is going to involve many asynchronous steps, latency is not a strong side of Nanoservices.

Evolutions

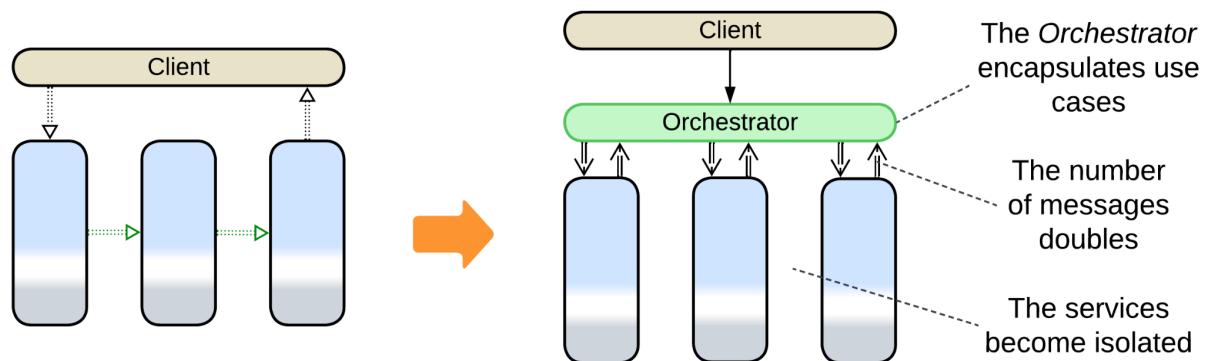
Pipeline [inherits its set of evolutions from Services](#). Filters can be added, split in two, merged, or replaced. Many systems employ a [Middleware](#) (a pub/sub or pipeline framework), a [Shared Repository](#) (which may be a database or a file system), or [Proxies](#).

There are a couple of pipeline-specific evolutions, with more details provided in [Appendix E](#):

- The first service of the *Pipeline* can be promoted to a [Front Controller](#) [SAHP] which tracks the status updates for every request it handles.



- Adding an [Orchestrator](#) turns a *Pipeline* into normal [Services](#). As the high-level business logic moves into the orchestration layer, the filters don't need to interact directly, therefore the inter-filter communication channels disappear and the system becomes identical to *Orchestrated Services*.



Summary

A *Pipeline* represents a data processing algorithm as a sequence of steps. It not only subdivides the system's code into smaller components but is also very flexible: its parts are

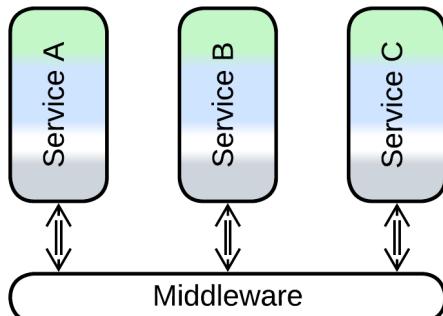
easy to add, remove, or replace. Multiple use cases can be built over the same set of services. Scalability is good. Event replay helps with debugging. However, its operational complexity restricts the architecture to smaller domains with a limited number of scenarios.

Part 3. Extension Metapatterns

These patterns extend Services, Shards, or even a *Monolith* with a layer that provides an aspect or two of the system's behavior and often glues other components together.

Middleware

The *Middleware* provides communication for the services



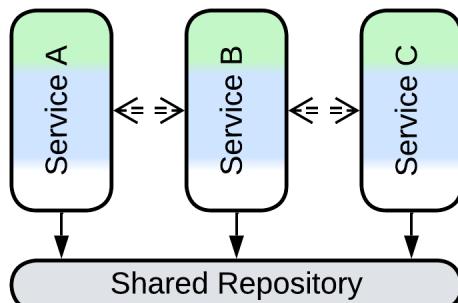
It may also manage their instances

Middleware is a layer that implements communication between instances of the system's components and it may also manage the instances. This way each instance is relieved of the need to track the other instances which it accesses.

Includes: (Message) Broker and Deployment Manager.

Shared Repository

The *Shared Repository* owns the system's data



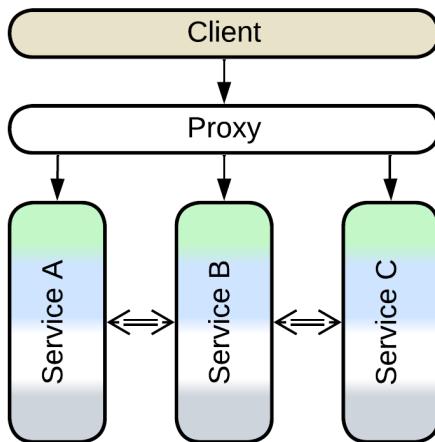
The services communicate directly or through the repository

A *Shared Repository* stores the system's data, maintains its integrity through transactions, and may support subscriptions to changes in subsets of the data. That lets other system components concentrate on implementing the business logic.

Includes: Shared Database, Blackboard, Data Grid or Space-Based Architecture, Shared Memory, and Shared File System.

Proxy

A *Proxy* stands between a (sub)system and its clients



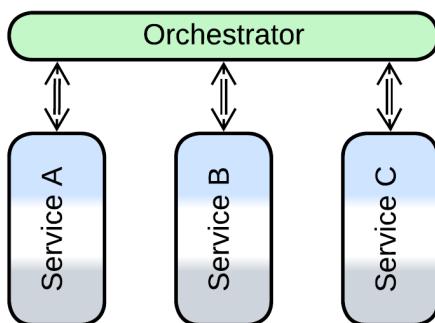
It implements a generic aspect of the system's behavior

A *Proxy* mediates between a system and its clients, transparently taking care of some generic functionality.

Includes: Full Proxy and Half-Proxy; Sidecar and Ambassador; Firewall, Response Cache, Load Balancer, Reverse Proxy and various Adapters, e.g. Anticorruption Layer, Open Host Service, XXX Abstraction Layers and Repository.

Orchestrator

The *Orchestrator* runs high-level scenarios by using the services



It also keeps the data of the services consistent

An *Orchestrator* implements use cases as sequences of calls to the underlying components which are usually left unaware of each other's existence.

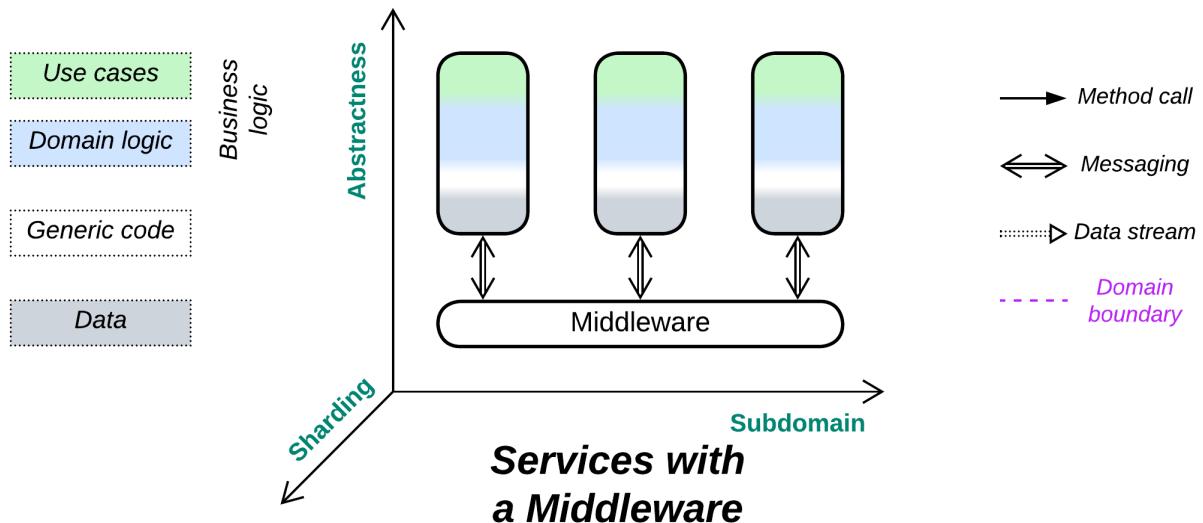
Includes: Workflow Owner, Application Layer, Facade, Mediator; API Composer, Scatter-Gather, MapReduce, Process Manager, Saga Execution Component, and Integration (Micro-)Service.

Combined Component

Several patterns combine the functionality of two or more extension layers.

Includes: Message Bus, API Gateway, Event Mediator, Enterprise Service Bus, Service Mesh, Middleware of Space-Based Architecture, and Shared Event Store.

Middleware



The line between disorder and order lies in logistics. Use a shared transport.

Known as: (Distributed) Middleware.

Aspects:

- Message Broker [[POSA1](#), [POSA4](#), [EIP](#), [MP](#)],
- Deployment Manager [[SAP](#), [FSA](#)].

Variants: Implementations differ in many dimensions.

The following combined patterns include *Middleware*:

- (with [Adapters](#)) Message Bus [[EIP](#)],
- (with [Proxies](#)) Service Mesh [[FSA](#), [MP](#)],
- (with an [Orchestrator](#)) Event Mediator [[FSA](#)],
- (with a [Shared Repository](#)) Persistent Event Log / Shared Event Store,
- (with an [Orchestrator](#) and [Adapters](#)) [Enterprise Service Bus](#) (ESB) [[FSA](#)].

Structure: A low-level layer that provides connectivity.

Type: Extension.

Benefits	Drawbacks
Separates connectivity concerns from the services that use it	May become a single point of failure
Transparent scaling of components	May increase latency
Available off the shelf	A generic <i>Middleware</i> may not fit specific communication needs

References: [[EIP](#)] has a lot of content on the implementation of messaging *Middleware*. [[POSA4](#)] features a chapter on *Middleware*. However, those books are old. [[DEDS](#)] is about Kafka, but it goes too far advertising it as a [Shared Event Store](#). There is also a [Wikipedia article](#).

Extracting transport into a separate layer relieves the components that implement business logic of the need to know the addresses and statuses of each other's instances. An industrial-grade, third-party *Middleware* is likely to be more stable and provide better error recovery than anything an average company can afford to implement on its own.

A *Middleware* may function as:

- A *Message Broker* [[POSA1](#), [POSA4](#), [EIP](#), [MP](#)] which provides a unified means of communication and implements some cross-cutting concerns like the persisting or logging of messages.
- A *Deployment Manager* [[SAP](#), [FSA](#)] which collects telemetry and manages service instances for error recovery and dynamic scaling.

As *Middleware* is ubiquitous and does not affect business logic, it is usually omitted from structural diagrams.

Performance

A *Middleware* may negatively affect performance when compared to direct communication between services. Old implementations (star topology) relied on a *Broker* [[POSA1](#), [POSA4](#), [EIP](#)] that used to add an extra network hop for each message and limited scalability. Newer *Mesh*-based variants avoid those drawbacks but are very complex and may have consistency issues (according to the [CAP theorem](#)).

A more subtle drawback is that transports supported or recommended by a *Middleware* may be suboptimal for some of the interactions in your system, causing programmers to hack around the limitations or build higher-level protocols on top of your *Middleware*. Both cases can be ameliorated by adding means for direct communication between the services to bypass the *Middleware* or by using multiple specialized kinds of *Middleware*. However, that adds to the complexity of the system – the very issue the *Middleware* promised to help with.

Dependencies

Each service depends both on the *Middleware* and on the API of every service it communicates with.



You may decide to use an [Anticorruption Layer](#) [[DDD](#)] over your *Middleware* just in case you may need to change its vendor in the future.

Applicability

Middleware helps:

- *Multi-component systems*. When a service has several instances deployed which need to be accessed, its clients must know the addresses of (or have channels to) its instances and use an algorithm for instance selection. As the number of services grows, so does the amount of information about the others that each of them needs to track. Even worse, services sometimes crash or are being redeployed, requiring complicated algorithms that queue messages to deliver them in order once the

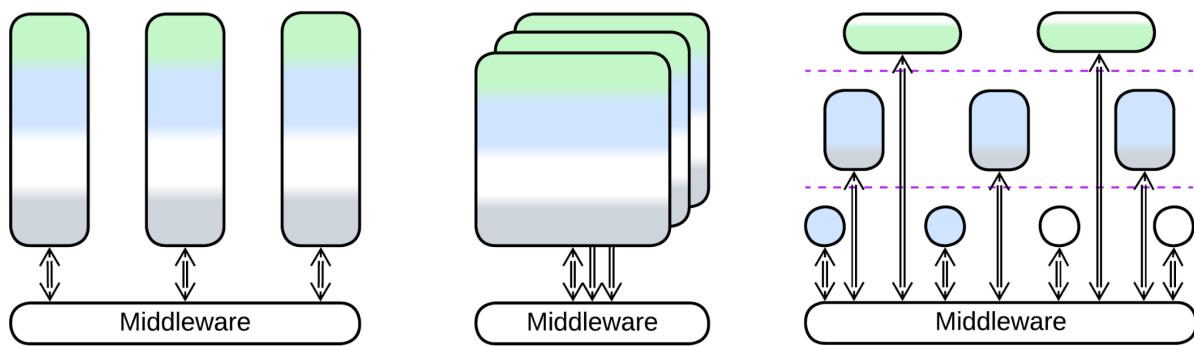
service returns to life. It makes all the sense to use a dedicated component that takes care of all of that.

- *Dynamic scaling*. It is good to have a single component that manages routes for interservice messaging to account for newly deployed (or destroyed) service instances.
- *Blue-green deployment, canary release, or dark launching (traffic mirroring)*. It is easier to switch, whether fully or in part, to a new version of a service when the communication is centralized.
- *System stability*. Most implementations of *Middleware* guarantee message delivery with unstable networks and failing components. Many persist messages to be able to recover from failures of the *Middleware* itself.

Middleware hurts:

- *Critical real-time paths*. An extra layer of message processing is bad for latency. Such messages may need to bypass the *Middleware*.

Relations



Middleware for Services

Middleware for Shards

Middleware for SOA

Middleware:

- Extends *Services*, *Service-Oriented Architecture*, or in rare cases, *Shards* or *Layers*.
- Can make a *Bus of Buses (Hierarchy)* or be merged with other extension metapatterns into several kinds of *Combined Components*.
- Is closely related to *Shared Repository*. A persistent *Middleware* employs a *Shared Repository*.
- Is usually implemented by a *Mesh* or *Microkernel* (which is often based on a *Mesh*).

Variants by functionality

There are several dimensions of freedom with a *Middleware*, some of them may be configurable:

By addressing (channels, actors, pub/sub, gossip)

Systems vary in the way their components address each other:

- *Channels* (one to one) – components which need to communicate establish a message channel between them. Once created, the channel accepts messages or a

data stream on its input end and transparently delivers them to its output end. Examples: sockets, Unix pipes, and private chats.

- *Actors* (many to one) – each component has a public mailbox. Any other component that knows its address or name may push a message into it. Examples: e-mail.
- *Publish/subscribe* (one to many) – a component can publish events to topics. Other components subscribe to those topics and one or all of the subscribers receive copies of a published event. Examples: IP multicast and subscriptions for e-mail notifications.
- *Gossip* (many to many) – an instance of a component periodically synchronizes its version of the system's state with random other instances, which further spread the updates. As time passes, more and more participants become aware of the changes in the system, leading to eventual consistency of the *replicas* of the system's state.

By flow (notifications, request/confirm, RPC)

Control flow may take one or more of the following approaches:

- *Notifications* – a component sends a message about an event that occurred and does not really care about the further consequences or wait for a response.
- *Request/confirm* – a component sends a message which requests that another component does something and sends back the result. The sender may execute other tasks meanwhile. A request usually includes a unique id that will be added to the confirmation for the *Middleware* to know which of the requests in progress the confirmation belongs to.
- *Remote procedure call* (RPC) is usually built on top of a request/confirm protocol. The difference is that the sender blocks on the *Middleware* while waiting for the confirmation, thus to the application code the whole process of sending the request and waiting for the confirmation looks like a single method call.

By delivery guarantee

If the transport (network) or the destination fails, a message may not be processed, or may be processed twice because of retries. A *Middleware* may [promise to deliver messages](#):

- *Exactly once*. This is the slowest case which is [implemented through distributed transactions](#). If the network, *Middleware*, or the message handler fails, there is no side effect, and the whole process of delivering and executing the message is repeated. The *exactly once* contract is used for financial systems where money should never disappear or duplicate during a transfer.
- *At least once*. On failure the message is redelivered, but the previous message could have already been processed (if only the confirmation was lost), thus there is a chance for a message to be processed twice. If the message is *idempotent* [MP], meaning that it sets a value ($x = 42$) instead of incrementing or decrementing it ($x = x + 2$), then we can more or less safely process it multiple times ($x = 42; x = 42; x = 42;$) and use the relatively fast *exactly once* guarantee.
- *At most once*. If anything fails, the message is lost and never retried. This is the fastest of the three guarantees, suitable for such monitoring applications as weather sensors – it is not too bad if a single temperature measurement disappears when you receive hundreds of them every day.
- *At will* (no guarantee). As with the bare [UDP transport](#), a message may disappear, become duplicated, or arrive out of order. That fits real-time streaming protocols

(video or audio calls) where it is acceptable to skip a frame while a frame coming too late is of no use at all. Each frame contains its sequence number, and it is up to the application to reorder and deduplicate the frames it receives.

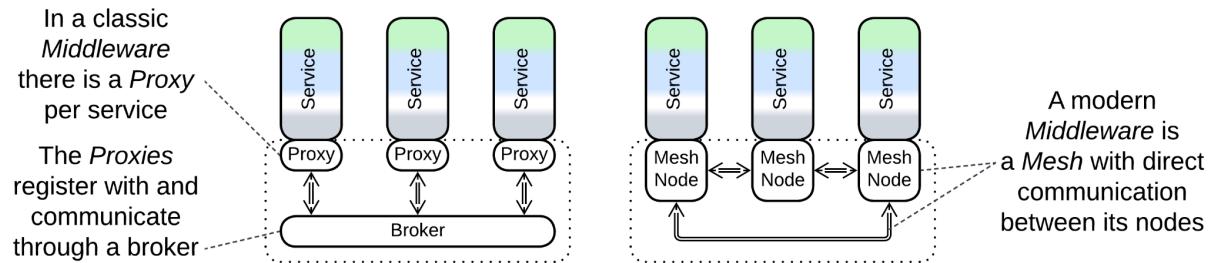
By persistence

A *Middleware* with a delivery guarantee needs to store messages whose delivery has not yet been confirmed. They may be:

- Written to a database of the *broker*.
- Persisted in a distributed database in brokerless (*Mesh*) systems.
- Replicated over an in-memory *Mesh* storage (like *Data Grid*).

If the messages are stored indefinitely, the *Middleware* becomes a *Persistent Event Log* or even a *Shared Event Store* with the schemas of the stored events coupling the involved services [*DEDS*] just like the database schema does in *Shared Repository*.

By structure (Microkernel, Mesh, Broker)



A *Middleware* may be:

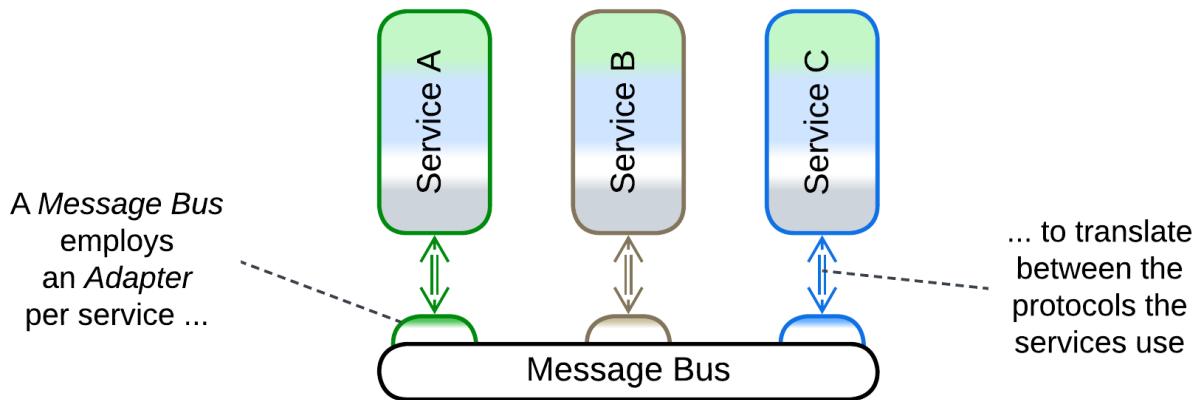
- Implemented by an underlying operating or virtualization system (see *Microkernel*).
- Run as a *Mesh* of identical modules co-deployed with the distributed components as *Sidecars*.
- Rely on a single *broker* [*POSA1*, *POSA4*, *EIP*] for coordination.

The last configuration is simpler but features a single point of failure unless multiple instances of the broker are deployed and kept synchronized.

Examples of merging Middleware and other metapatterns

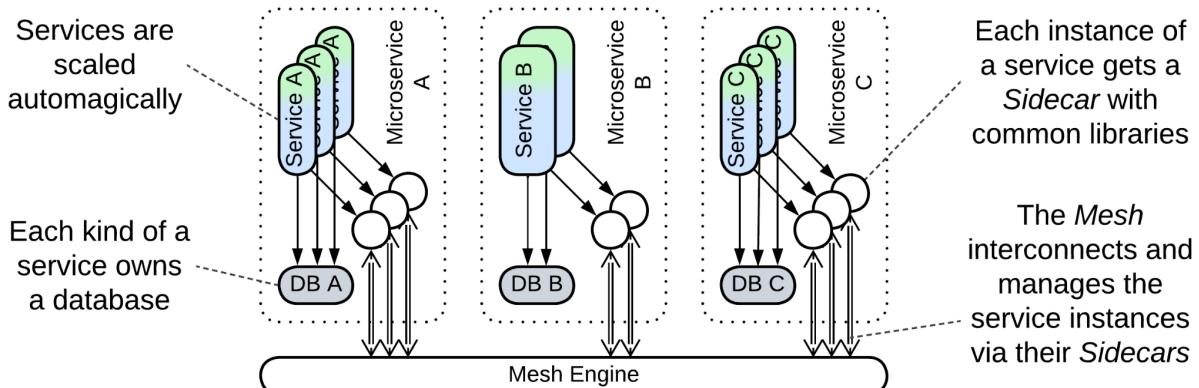
There are several patterns which extend *Middleware* with other functions:

Message Bus



A *Message Bus* [EIP] employs one or more *Adapters* per service to let the services intercommunicate even if they differ in protocols. That helps to integrate legacy services without much change to their code but it degrades overall performance as up to two protocol translations per message are involved.

Service Mesh



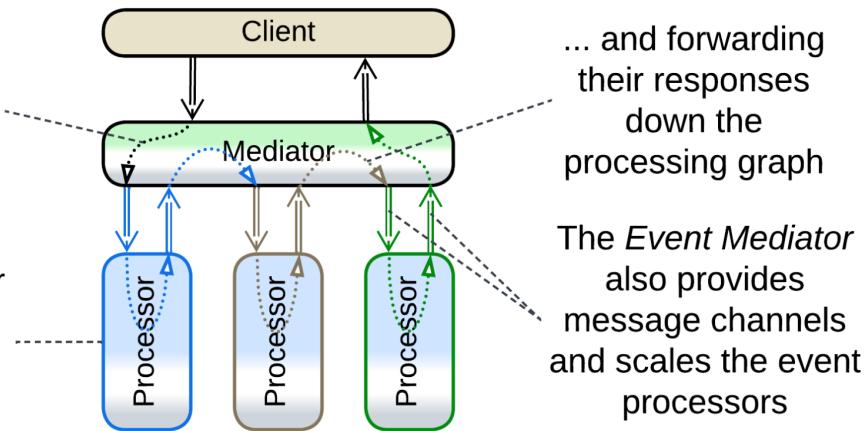
Service Mesh [FSA, MP] is a smart *Mesh*-based *Middleware* that manages service instances and employs at least one co-located *Proxy* (called *Sidecar* [DDS]) per service instance deployed. The *Sidecars* may provide protocol translation and cover cross-cutting concerns such as encryption or logging. They make a good place for shared libraries.

The internals of *Service Mesh* are discussed in the *Mesh chapter*.

Event Mediator

The *Event Mediator* builds a *Pipeline* by transferring client requests to event processors ...

Each event processor contains business logic but does not know the event flow



... and forwarding their responses down the processing graph

The *Event Mediator* also provides message channels and scales the event processors

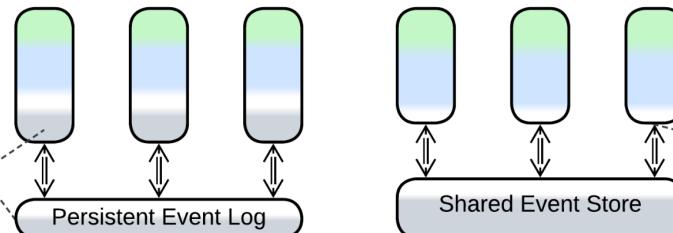
Event Mediator [FSA], which pervades both [Event-Driven Architectures](#) and [Nanoservices](#), melds a *Middleware* (used for delivery of messages) and an *Orchestrator* (that coordinates high-level use cases). A message arrives to a service and is responded to without any explicit component on the service's side – it appears *out of thin Middleware* which implements the entire integration logic.

Slightly more details on the *Event Mediator* are [provided in the *Orchestrator* chapter](#).

Persistent Event Log, Shared Event Store

A *Middleware* may persist messages ...

Each service owns its state

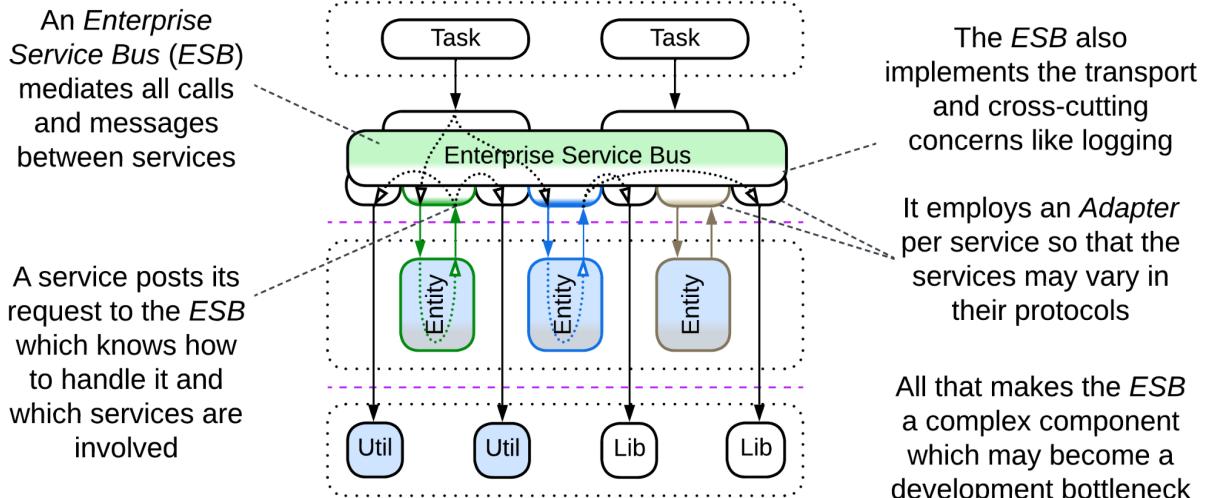


... or it may also store the history of the services' states

When a *Middleware* persists messages, it takes on the function (and drawbacks) of a [Shared Repository](#). A *Persistent Event Log* allows to replay incoming events for a given service (to help a debug session or fix data corrupted by a bug) while a *Shared Event Store* also captures changes of the internal states of the services [DEDS], [replacing their private databases](#). However, with either approach, changing an event field impacts all the services that use the event and may involve rewriting the entire event log (system's history) [DEDS].

This pattern is detailed in the [Combined Component](#) chapter.

Enterprise Service Bus (ESB)



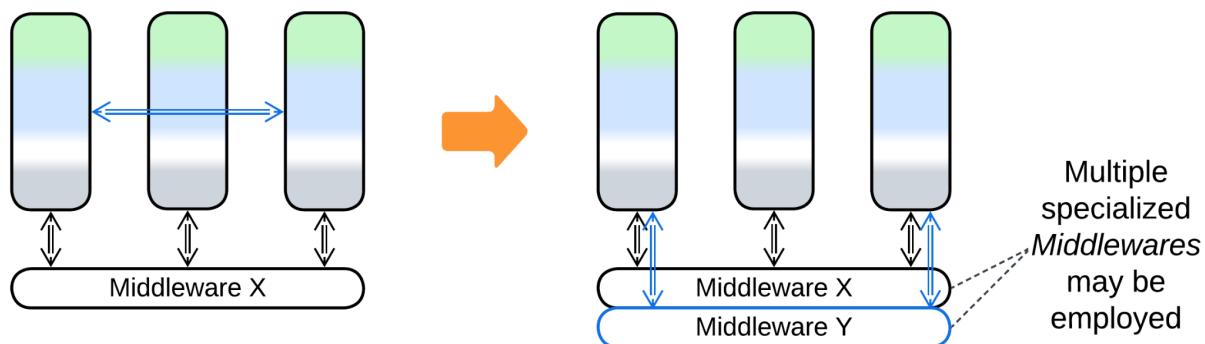
[Enterprise Service Bus \(ESB\)](#) [[ESB](#)] is a mixture of *Message Bus* and *Event Mediator*. A *ESB* blends a *Middleware* and an [Orchestrator](#) and adds an [Adapter](#) per service as a topping. It emerged to connect components that originated in incompatible networks of organizations that had been acquired by a corporation.

See the [chapter about Service-Oriented Architecture](#).

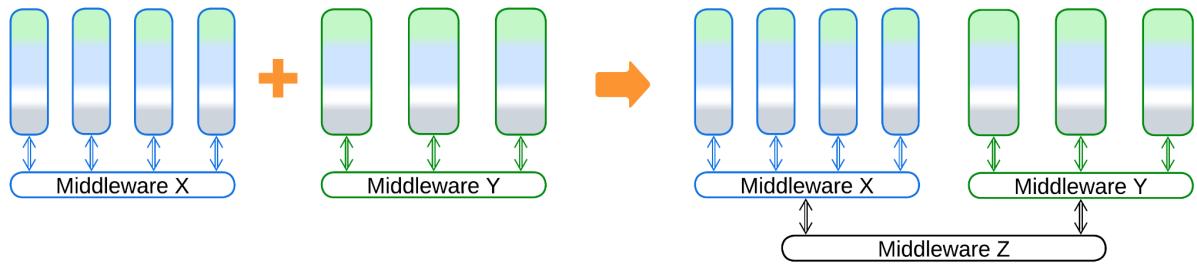
Evolutions

A *Middleware* is unlikely to be removed (though it may be replaced) once it is built into a system. There are few evolutions for *Middleware* because it is usually a third-party product and thus unlikely to be modified in-house:

- If the *Middleware* in use does not fit the preferred mode of communication between some of your services, there is the option to deploy a second, specialized *Middleware*.



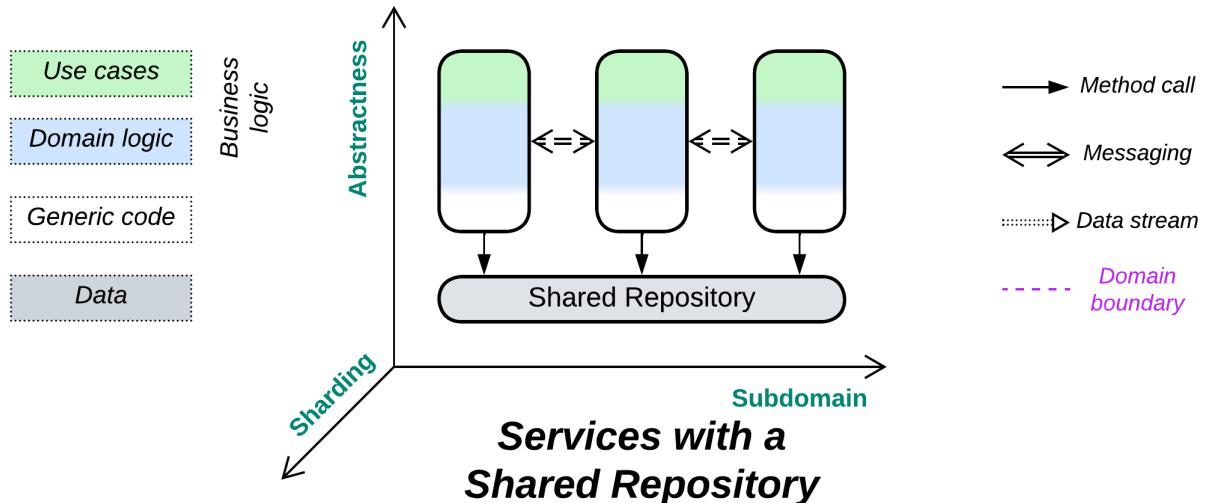
- If several existing systems need to be merged, that is accomplished by adding yet another layer of *Middleware*, resulting in a [Bottom-up Hierarchy \(Bus of Buses\)](#).



Summary

A *Middleware* is a ready-to-use component that provides a system of services with means of communication, scalability, and error recovery. It is very common in distributed backends.

Shared Repository



Knowledge itself is power. Sharing data is simple (& stupid).

Known as: Shared Repository [[POSA4](#)].

Aspects:

- Data storage,
- Data consistency,
- Data change notifications.

Variants:

- Shared Database [[EIP](#)] / [Integration Database](#) / Data Domain [[SAHP](#)] / Database of Service-Based Architecture [[FSA](#)],
- [Blackboard](#) [[POSA1](#), [POSA4](#)],
- Data Grid of [Space-Based Architecture](#) [[SAP](#), [FSA](#)] / Replicated Cache [[SAHP](#)] / Distributed Cache,
- Shared Memory,
- Shared File System,
- (with a [Middleware](#)) Persistent Event Log / Shared Event Store,
- (inexact) Stamp Coupling [[SAHP](#)].

Structure: A layer of data shared among higher-level components.

Type: Extension for [Services](#) or [Shards](#).

Benefits	Drawbacks
Supports domains with coupled data	A single point of failure
Implements data access and synchronization (consistency) concerns	All the services depend on the schema of the shared data
Helps saving on hardware, licenses, traffic, and administration	A single database technology may not fit the needs of all the services equally well
Quick start for a project	Limits scalability

References: [[DDIA](#)] is all about databases; [[FSA](#)] has chapters on *Service-Based Architecture* and *Space-Based Architecture*; [[DEDS](#)] deals with *Shared Event Store*.

A *Shared Repository* builds communication in the system around its data, which is natural for [data-centric domains](#) and multiple [instances of a stateless service](#) and may often

simplify development of a system of [Services](#) that need to exchange data. It covers the following concerns:

- Storage of the entire domain data.
- Keeping the data self-consistent by providing atomic transactions for use by the application code.
- Communication between the services (if the repository supports notifications on data change).

The drawbacks are extensive coupling (it's hard to alter a thing which is used in many places throughout the entire system) and limited scalability (even distributed databases struggle against distributed locks and the need to keep their nodes' data in sync).

Performance

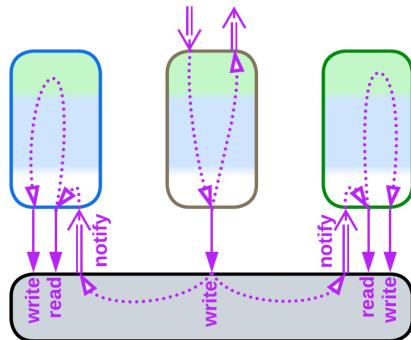
A shared database with consistency guarantees ([ACID](#)) is likely to lower the total resource consumption compared to one database per service (as the services don't need to implement and keep updated [CQRS views](#) [[DDIA](#), [MP](#)] of other services' data) but it increases latency and it may become the system's performance bottleneck. Moreover, by using a shared database services lose the ability to choose the database technologies which best fit their tasks and data.

Another danger lies with locking records inside the database. Different services may use different order of tables in transactions, hitting deadlocks in the database engine which show up as transaction timeouts.

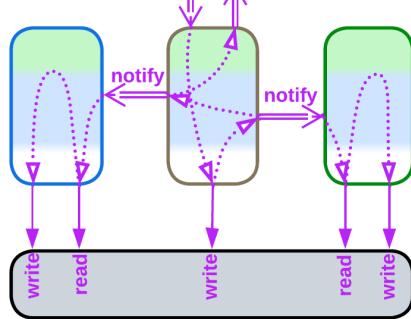
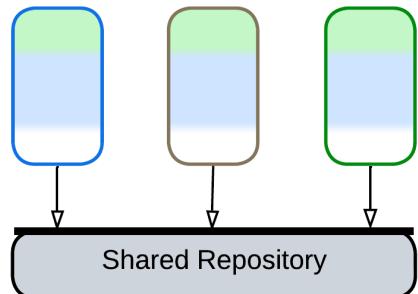
Non-transactional distributed databases may be very fast when colocated with the services (see [Space-Based Architecture](#)) but the resource consumption becomes very high because of the associated data duplication (as every instance of each service gets a copy of the entire dataset) and simultaneous writes may corrupt the data (cause inconsistencies or merge conflicts).

Dependencies

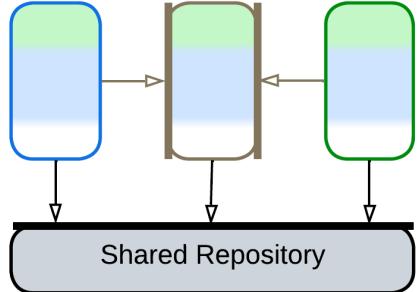
Normally, every service depends on the repository. If the repository does not provide notifications on changes to the data, the services may need to communicate directly, in which case they will also depend on each other through [choreography](#) or [mutual orchestration](#).



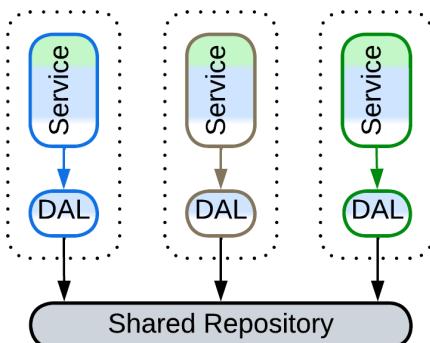
If the repository supports notifications, the services are independent



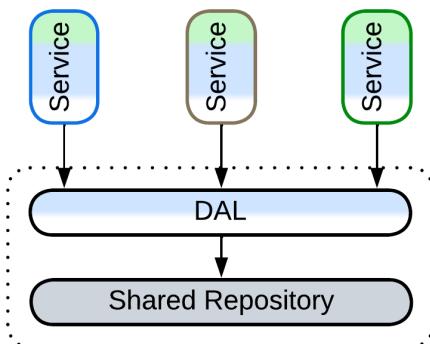
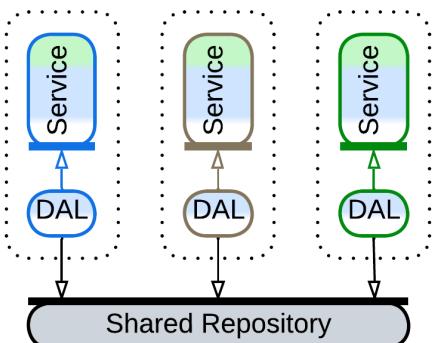
Otherwise a service depends on the services it subscribes to



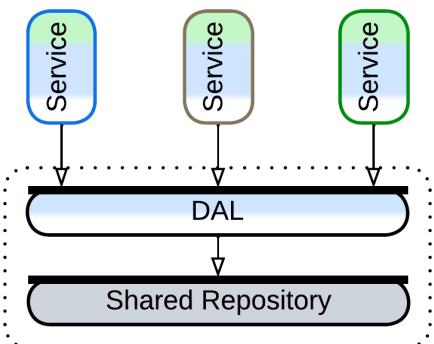
The dependency on repository technology and a data schema is dangerous for long-running projects as both of them may need to change sooner or later. Decoupling the code from the data storage is done with [yet another layer of indirection](#) which is called a [Database Abstraction Layer](#) (DAL), a Database Access Layer [POSA4], or a Data Mapper [PEAA]. The DAL, which translates between the data schema and database's API on one side and the business logic's SPI on the other side, may reside inside each service or wrap the database:



Each service may have its own database abstraction layer (DAL)



Or there may be a system-wide DAL co-located with the database



Still, the DAL does not remove shared dependencies and only adds some flexibility. It seems that there is a peculiar kind of coupling through shared components: if one of the services needs to change the database schema or technology to better suit its needs, it is

unable to do so because other components rely on (and exploit) the old schema and technology. Even deploying a second database, private to the service, is often not an option, as there is no convenient way to keep the databases in sync.

Applicability

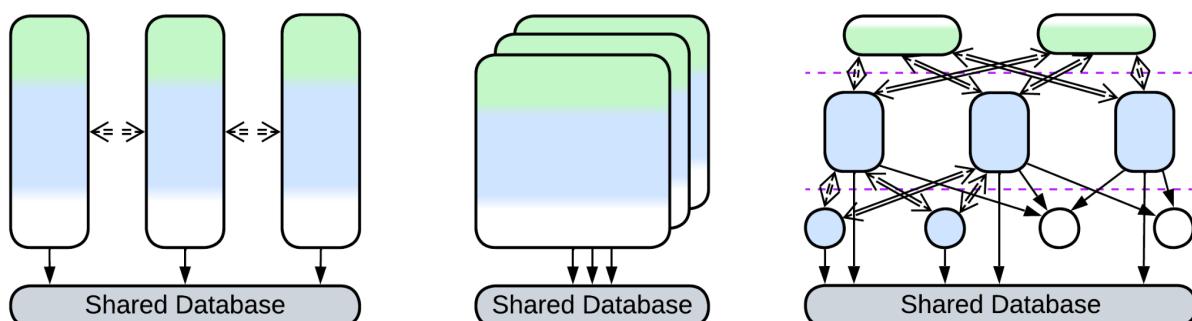
Shared Repository is good for:

- *Data-centric domains*. If most of your domain's data is used in every subdomain, keeping any part of it private to a single service will be a pain in the system design. Examples include a [ticket reservation system](#) and even the minesweeper game.
- A *scalable service*. When you run several [instances](#) of a service, like in [Microservices](#), the instances are likely to be identical and stateless, with the service's data pushed out to a database shared among the instances.
- *Huge datasets*. Sometimes you may need to deal with a lot of data. It is unwise (meaning expensive) to stream and replicate it between your services just for the sake of ensuring their isolation. Share it instead. If the data does not fit in an ordinary database, some kind of [Space-Based Architecture](#) (which [was invented to this end](#)) may become your friend.
- *Quick simple projects*. Don't over-engineer if the project won't live long enough to benefit from its flexibility. You may also save a buck or two on the storage.

Shared Repository is bad for:

- *Quickly evolving, complex projects*. As everything changes, you just cannot devise a stable schema, while every change of the database schema breaks all the services.
- *Varied forces and algorithms*. Different services may require different kinds of databases to work efficiently.
- *Big data with random writes*. Your data does not fit on a single server. If you want to avoid write conflicts, you must keep all the database nodes synchronized, which kills performance. If you let them all broadcast their changes asynchronously, you get collisions. You may want to first decouple and [shard](#) the data as much as possible, and then turn your attention to esoteric databases, specialized caches, and even tailor-made [Middleware](#) to get out of the trouble.

Relations



**Shared Repository
for Services**

**Shared Repository
for Shards**

**Shared Repository
for SOA**

Shared Repository:

- Extends [Services](#), [Service-Oriented Architecture](#), [Shards](#), or occasionally [Layers](#).
- Is a part of a [persistent Middleware](#) or [Nanoservices](#).

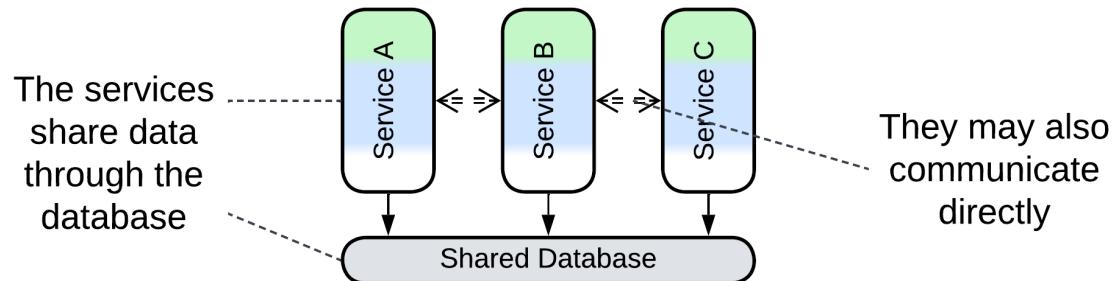
- Is [closely related](#) to [Middleware](#).
- May be implemented by a [Mesh](#).

Variants

Shared Repository is a sibling of [Middleware](#). While a *Middleware* assists direct communication between services (*shared-nothing* messaging), a *Shared Repository* grants them indirect communication through access to an external state (similar to *shared memory*) which usually stores all the data for the domain.

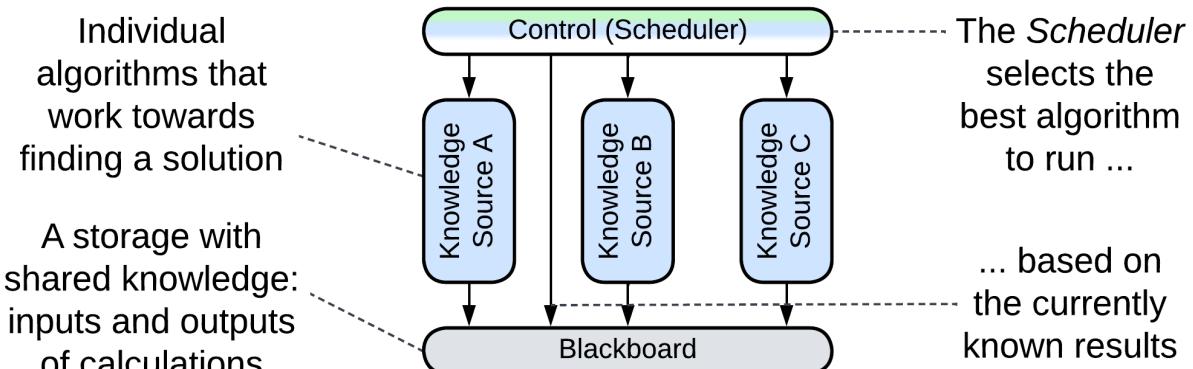
A *Shared Repository* may provide a generic interface (e.g. SQL) or a custom API (with a domain-aware [Adapter](#) / [ORM](#) for the database). The *repository* can be anything ranging from a trivial OS file system or a memory block accessible from all the components to an ordinary database to a [Mesh](#)-based, distributed [tuple space](#):

Shared Database, Integration Database, Data Domain, Database of Service-Based Architecture



Shared Database [EIP], [Integration Database](#), or [Data Domain](#) [SAHP] is a single database available to several [services](#). The services may subscribe to data change triggers in the database itself or notify each other directly about domain events. The latter is often the case with [Service-Based Architecture](#) [FSA] which consists of large services dedicated to subdomains.

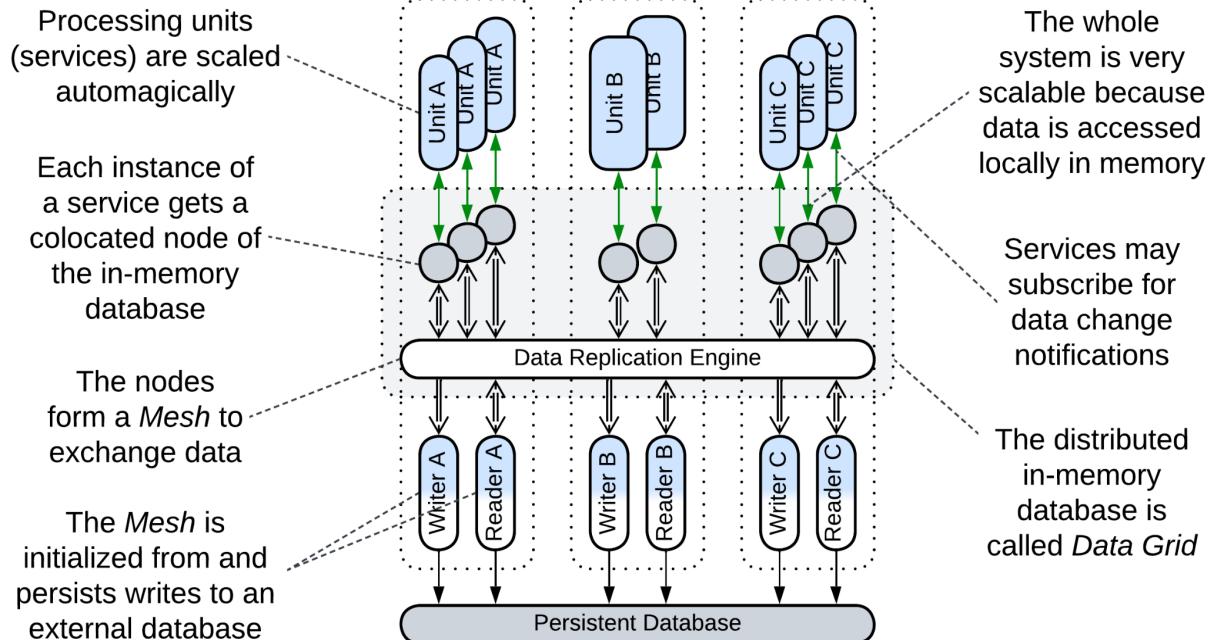
Blackboard



[Blackboard](#) [POSA1, POSA4] was used for non-deterministic calculations where several algorithms were concurring and collaborating to gradually build a solution from incomplete inputs. The *control* ([Orchestrator](#)) component schedules the work of several *knowledge sources* ([Services](#)) which encapsulate algorithms for processing the data stored in the *blackboard* (*Shared Repository*). This approach has likely been superseded by convolutional neural networks.

Examples: several use cases are [mentioned on Wikipedia](#).

Data Grid of Space-Based Architecture (SBA), Replicated Cache, Distributed Cache



The [Space-Based Architecture](#) (SBA) [[SAP](#), [FSA](#)] is a [Service Mesh](#) (a [Mesh](#)-based [Middleware](#) with at least one [Proxy](#) per service instance) which also implements an in-memory [tuple space](#) (shared dictionary). Although it does not provide a full-featured database interface it has very good performance, elasticity, and fault tolerance, while some implementations allow for dealing with datasets which are much larger than anything digestible by ordinary databases. Its drawbacks include write collisions and high operating costs (huge traffic for data replication and lots of RAM to store the replicas).

The main components of the architecture are:

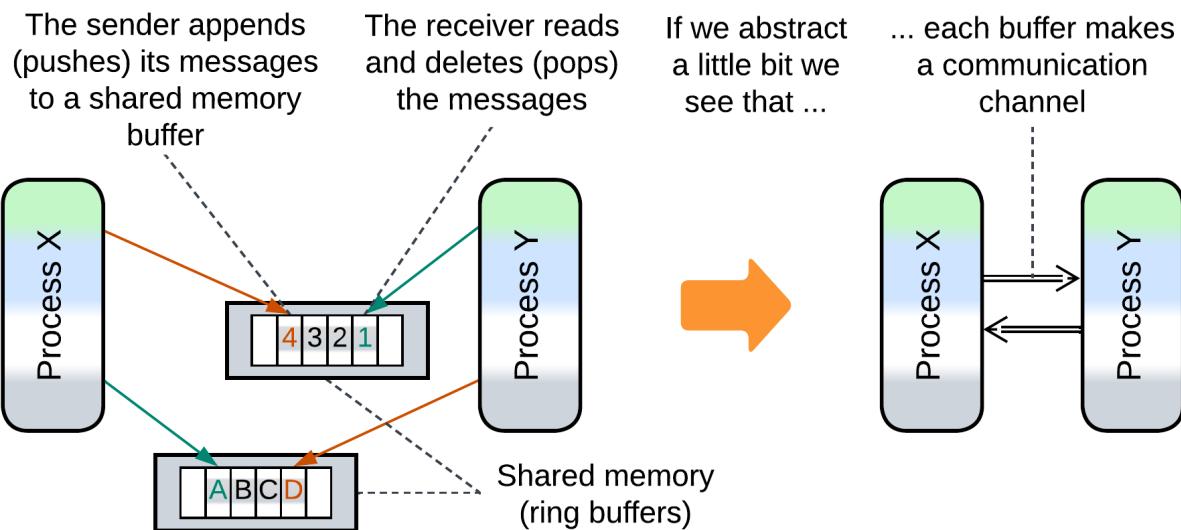
- *Processing Units* – the [Services](#) with the business logic. There may be one class of *Processing Units*, making SBA look like [Pool \(Shards\)](#), or multiple classes, in which case the architecture becomes similar to [Microservices](#) with a [Shared Database](#).
- *Data Grid (Replicated Cache [SAHP])* – a [Mesh](#)-based in-memory database. Each node of the *Data Grid* is co-located with a single instance of a *Processing Unit*, providing the latter with very fast access to the data. Changes to the data are replicated across the grid by its virtual *Data Replication Engine* which is usually implemented by every node of the grid.
- *Persistent Database* – an external database which the *Data Grid* replicates (caches). Its schema is encapsulated in the *Readers* and *Writers*.
- *Data Readers* – components that read any data unavailable in the *Data Grid* from the *Persistent Database*. Most cases see *Readers* employed upon starting the system to upload the entire contents of the database into the memory of the nodes.
- *Data Writers* – components that replicate the changes done in the *Data Grid* to the persistent storage to assure that no updates are lost if the system is shut down. There can be a pair of *Reader* and *Writer* per class of *Processing Units* (subdomain) or a global pair that processes all read and write requests.

SBA provides nearly perfect scalability (high read and write throughput as all the data is [cached](#)) and elasticity (new instances of *Processing Units* are created and initialized very quickly as they copy their data from already running units with no need to access the *Persistent Database*). Though for smaller datasets the entire database is [replicated](#) to every node of the grid (*Replicated Cache mode*), Space-Based Architecture also allows the processing of datasets that don't fit into the memory of a single node by assigning each node a [shard](#) of the dataset (*Distributed Cache mode*).

The drawbacks of this architecture include:

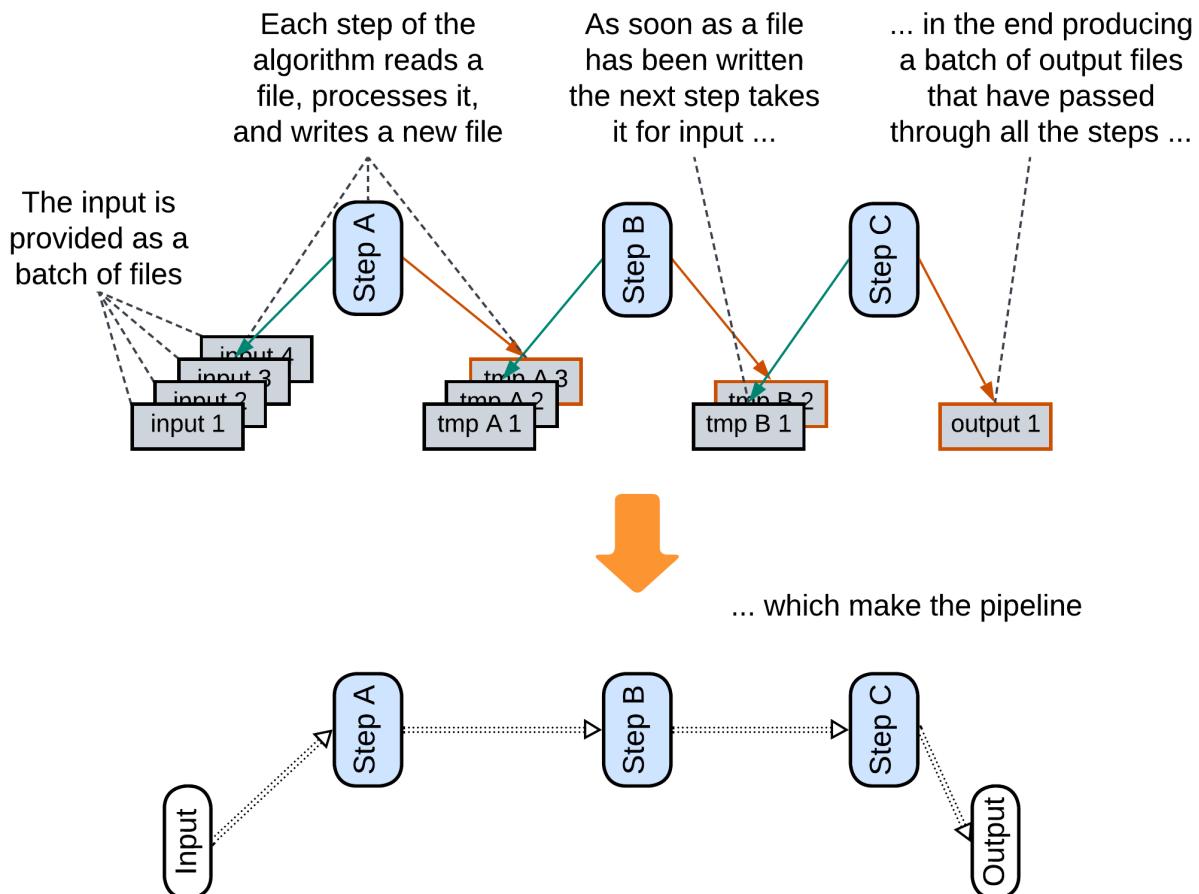
- Structural and operational complexity.
- Very basic dictionary-like interface of the *tuple space* (no joins or other complex operations).
- High traffic for data replication among the nodes.
- Data collisions if multiple clients change the same piece of data simultaneously.

Shared Memory



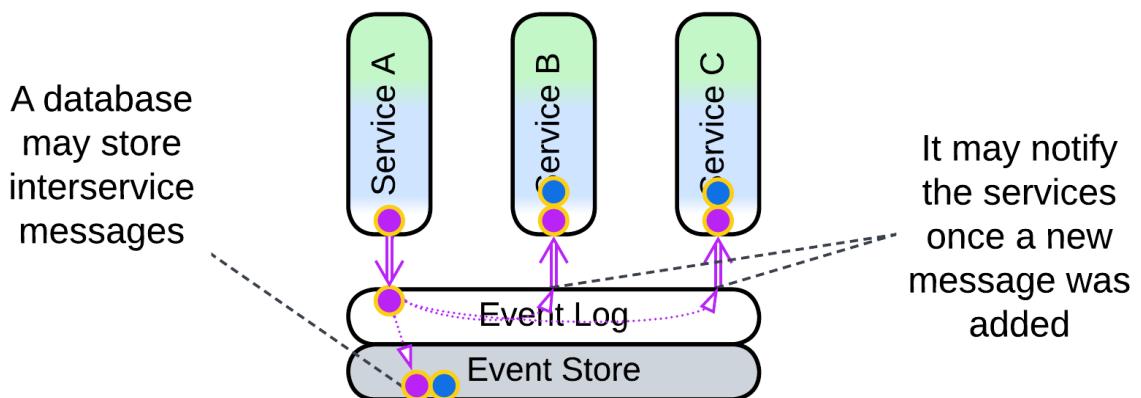
Several actors (processes, modules, device drivers) communicate through one or more mutually accessible data structures (arrays, trees, or dictionaries). Accessing a shared object may require some kind of synchronization (e.g. taking a *mutex*) or employ [atomic variables](#). Notwithstanding that communication via *shared memory* is the archenemy of ([shared-nothing](#)) messaging it is actually used to implement messaging: high-load multi-process systems (web browsers and high-frequency trading) rely on shared memory *mailboxes* for messaging between their [constituent processes](#).

Shared File System



As a file system is a kind of shared dictionary, writing and reading files can be used to transfer data between applications. A [data processing Pipeline](#) which stores intermediate results in files benefits from the ability to restart its calculation from the last successful step because files are persistent [[DDIA](#)].

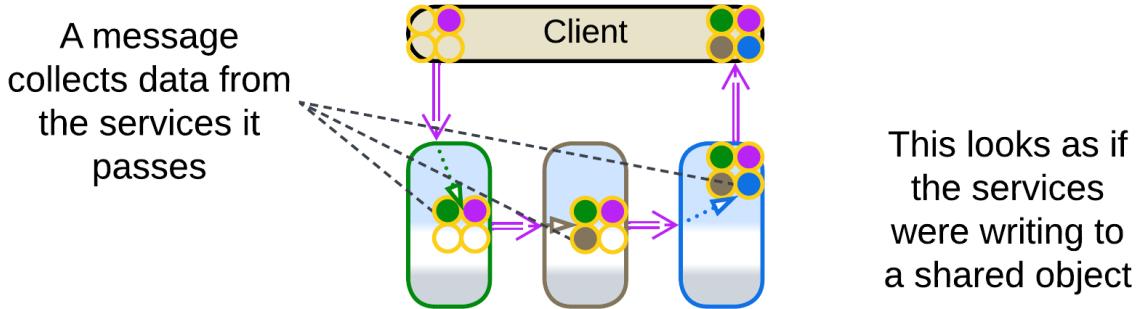
Persistent Event Log, Shared Event Store



A database which stores events (event log for interservice events, event store for internal state changes) can be used as a [Middleware](#): an event producer writes its events to a topic in the repository while event consumers get notified as soon as a new record appears.

More details are [available](#) in the *Combined Component* chapter.

(inexact) Stamp Coupling



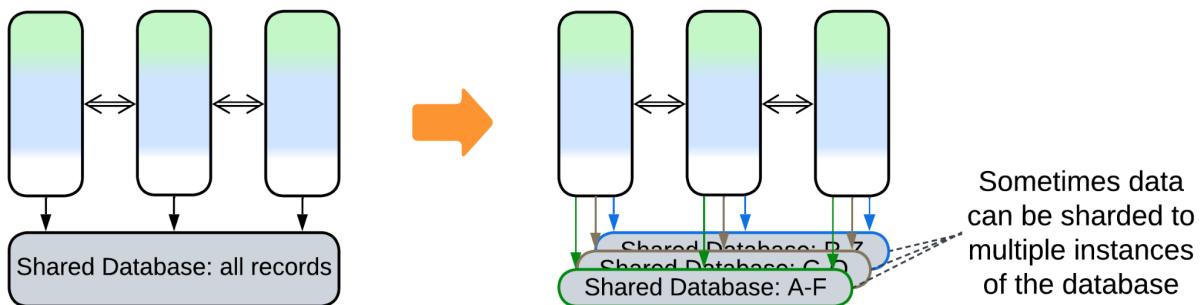
Stamp Coupling [SAHP] happens when a single data structure passes through an entire [Pipeline](#), with separate fields of the data structure targeting individual processing steps.

A [choreographed](#) system with no shared databases does not provide any way to aggregate the data spread over its multiple services. If we need to collect everything known about a user or purchase, we pass a query message through the system, and every service appends to it whatever it knows of the subject (just like post offices add their *stamps* to a letter). The unified message becomes a kind of virtual (temporary) *Shared Repository* which the services (*Content Enrichers* according to [EIP]) write to. This also manifests in the dependencies: all the services [depend on the format of the query message](#) as they would on the schema of a *Shared Repository*, instead of depending on each other, as is usual with *Pipelines*.

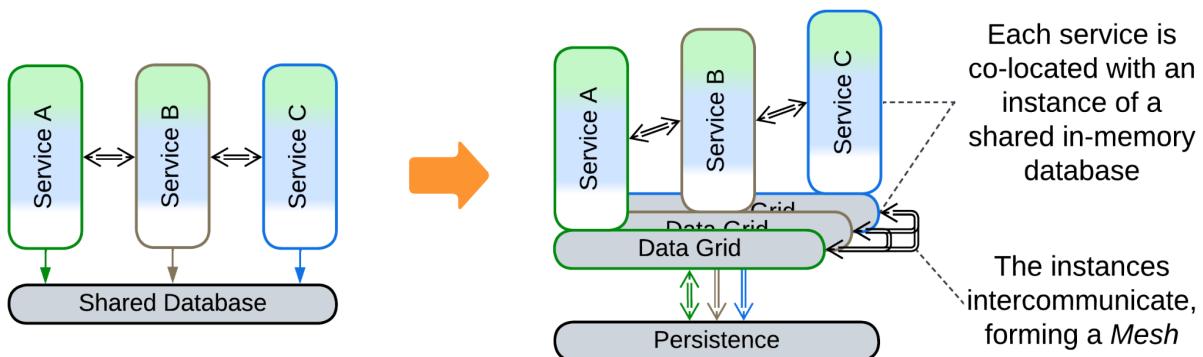
Evolutions

Once a database appears, it is unlikely to go away. I see the following evolutions to improve performance of the data layer:

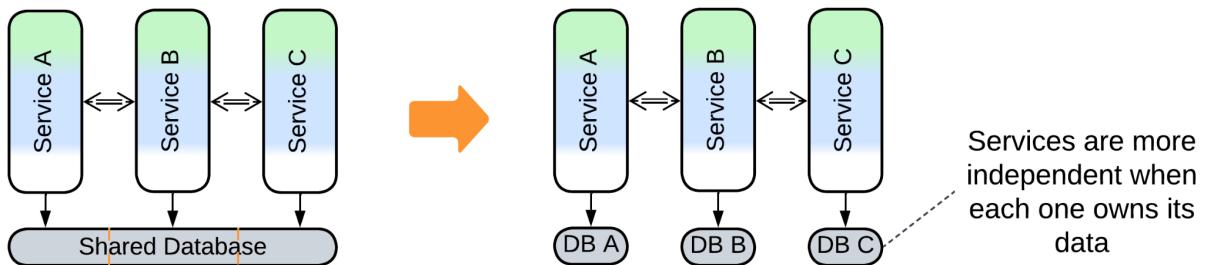
- [Shard](#) the database.



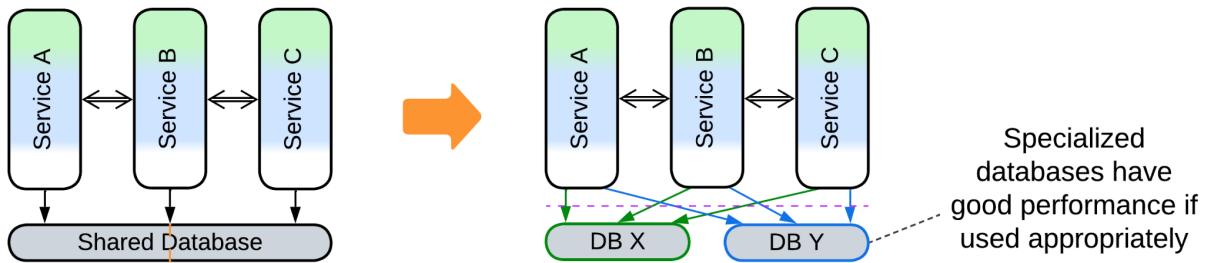
- Use [Space-Based Architecture](#) for dynamic scalability.



- Divide the data into private databases.



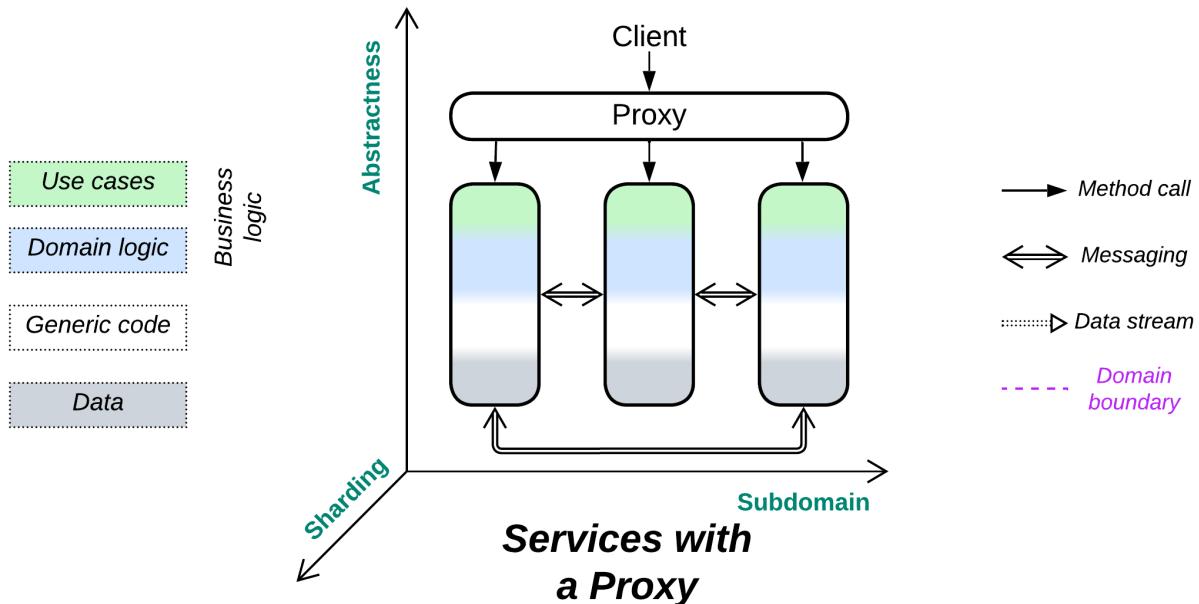
- Deploy specialized databases ([Polyglot Persistence](#)).



Summary

A *Shared Repository* encapsulates a system's data, allowing for [data-centric](#) development and kickstarting [Service-Based](#) architectures through simplifying interservice interactions. Its downsides are a frozen data schema and limited performance.

Proxy



Should I build the wall? A layer of indirection between your system and its clients.

Known as: Proxy [[GoF](#)].

Aspects:

- Routing,
- Offloading.

Variants:

By [transparency](#):

- Full Proxy,
- Half-Proxy.

By placement:

- Separate deployment,
- On the system side: Sidecar [[DDS](#)],
- On the client side: Ambassador [[DDS](#)].

By function:

- Firewall / [\(API\) Rate Limiter](#) / API Throttling,
- Response Cache / [Read-Through Cache](#) / [Write-Through Cache](#) / [Write-Behind Cache](#) / Cache [[DDS](#)] / Caching Layer [[DDS](#)] / Distributed Cache / Replicated Cache,
- [Load Balancer](#) [[DDS](#)] / Sharding Proxy [[DDS](#)] / [Cell Router](#) / Messaging Grid [[FSA](#)] / Scheduler,
- Dispatcher [[POSA1](#)] / [Reverse Proxy](#) / [Ingress Controller](#) / [Edge Service](#) / [Microgateway](#),
- Adapter [[GoF](#), [DDS](#)] / Anticorruption Layer [[DDD](#)] / Open Host Service [[DDD](#)] / Gateway [[PEAA](#)] / Message Translator [[EIP](#), [POSA4](#)] / API Service / Cell Gateway / (inexact) Backend for Frontend / [Hardware Abstraction Layer](#) (HAL) / [Operating System Abstraction Layer](#) (OSAL) / Platform Abstraction Layer (PAL) / [Database Abstraction Layer](#) (DBAL or DAL) / Database Access Layer [[POSA4](#)] / Data Mapper [[PEAA](#)] / Repository [[PEAA](#), [DDD](#)].
- (with [Orchestrator](#)) API Gateway [[MP](#)].

See also [Backends for Frontends](#) (a Gateway per client type).

Structure: A layer that pre-processes and/or routes user requests.

Type: Extension.

Benefits	Drawbacks
Separates cross-cutting concerns from the services	A single point of failure
Decouples the system from its clients	Most proxies degrade latency
Low attack surface	
Available off the shelf	

References: Half of [\[DDS\]](#) is about the use of *Proxies*. See also: [\[POSA4\]](#) on *Proxy*; [Chris Richardson](#) and [Microsoft](#) on *API Gateway*; [Martin Fowler](#) on *Gateway*, *Facade* and *API Gateway*.

A *Proxy* stands between a (sub)system's implementation and its users. It receives a request from a client, does some pre-processing, then forwards the request to a lower-level component. In other words, a *Proxy* encapsulates selected aspects of the system's communication with its clients by serving as yet another layer of indirection. It may also decouple the system's internals from changes in the public protocol. The [main functions](#) of a proxy include:

- *Routing* – a *Proxy* tracks addresses of deployed instances of the system's components and is able to forward a client's request to the [shard](#) or [service](#) which can handle it. Clients need to know only the public address of the *Proxy*. A *Proxy* may also respond on its own if the request is invalid or there is a matching response in the *Proxy*'s cache.
- *Offloading* – a *Proxy* may implement generic aspects ([cross-cutting concerns](#)) of the system's public interface, such as authentication, authorisation, encryption, request logging, web protocol support, etc. which would otherwise need to be implemented by the underlying system components. That allows for the services to concentrate on what you write them for – the business logic.

Performance

Most kinds of proxies trade latency (the extra network hop) for some other quality:

- A [Firewall](#) slows down processing of good requests but *protects* the system from attacks.
- Both a [Load Balancer](#) and a [Dispatcher](#) allow for the use of multiple servers (with identical or specialized components, correspondingly) to improve the system's *throughput* but they still add to the minimum latency.
- An [Adapter](#) adds *compatibility* but its latency cost is higher than with other *Proxies* as it not only forwards the original message but also changes its payload – an activity which involves data processing and serialization.

A [Cache](#) is a bit weird in that respect. It improves latency and throughput for repeated requests but degrades latency for unique ones. Furthermore, it is often colocated with some other kind of *Proxy* to avoid the extra network hop between the *Proxies*, which makes caching almost free in terms of latency.

Dependencies

Proxies widely vary in their functionality and level of intrusiveness. The most generic proxies, like *Firewalls*, may not know anything about the system or its clients. A *Response Cache* or *Adapter* must parse incoming messages, thus it depends on the communication protocol and message format. A *Load Balancer* or *Dispatcher* is aware of both the protocol and system composition.

In fact, because Proxies tend to have their dependencies configured on startup or through their APIs, there is no need to modify the code of a Proxy each time something changes in the underlying system.

Applicability

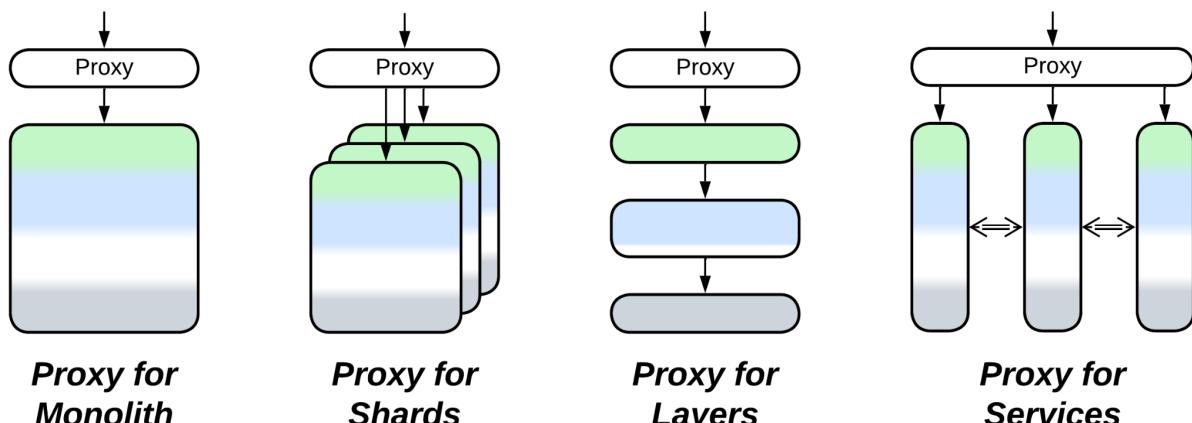
Proxy helps with:

- *Multi-component systems*. Having multiple types and/or instances of services means there is a need to know the components' addresses to access them. A *Proxy* encapsulates that knowledge and may also provide other common functionality as an extra benefit.
- *Dynamic scaling or sharding*. The *Proxy* both knows the system's structure (the address of each instance of a service) and delivers user requests, thus it is the place to implement *sharding* (when a service instance is [dedicated to a subset of users](#)) or *load balancing* (when any service instance can [serve any user](#)) and even manage the size of the service instance *Pool*.
- *Multiple client protocols*. When the *Proxy* is the endpoint for the system's users it may translate multiple external (user-facing) protocols into a unified internal representation.
- *System security*. Though a *Proxy* does not make a system more secure, it takes away the burden of security considerations from the services which implement the business logic, improving the separation of concerns and making the system components more simple and stupid. An off-the-shelf *Proxy* may be less vulnerable compared to in-house services (but don't disregard [security through obscurity!](#)).

Proxy hurts:

- *Critical real-time paths*. It adds an extra hop in request processing, increasing latency for thoroughly optimized use cases. Such requests may need to bypass *Proxies*.

Relations



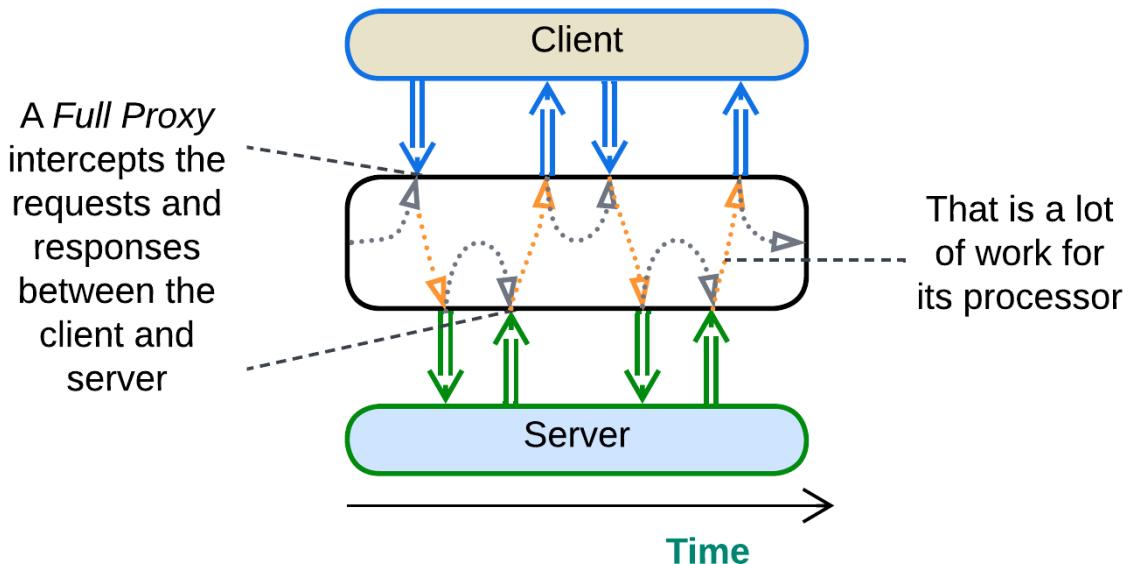
Proxy:

- Extends [Monolith](#) or [Layers](#) (forming [Layers](#)), [Shards](#), or [Services](#).
- Can be extended by another *Proxy* or merged with an [Orchestrator](#) into an [API Gateway](#).
- At least one *Proxy* per [service](#) is employed by [Message Bus](#), [Enterprise Service Bus](#), [Service Mesh](#), and [Hexagonal Architecture](#).
- Is a special case (when there is a single kind of client) of [Backends for Frontends](#).

Variants by transparency

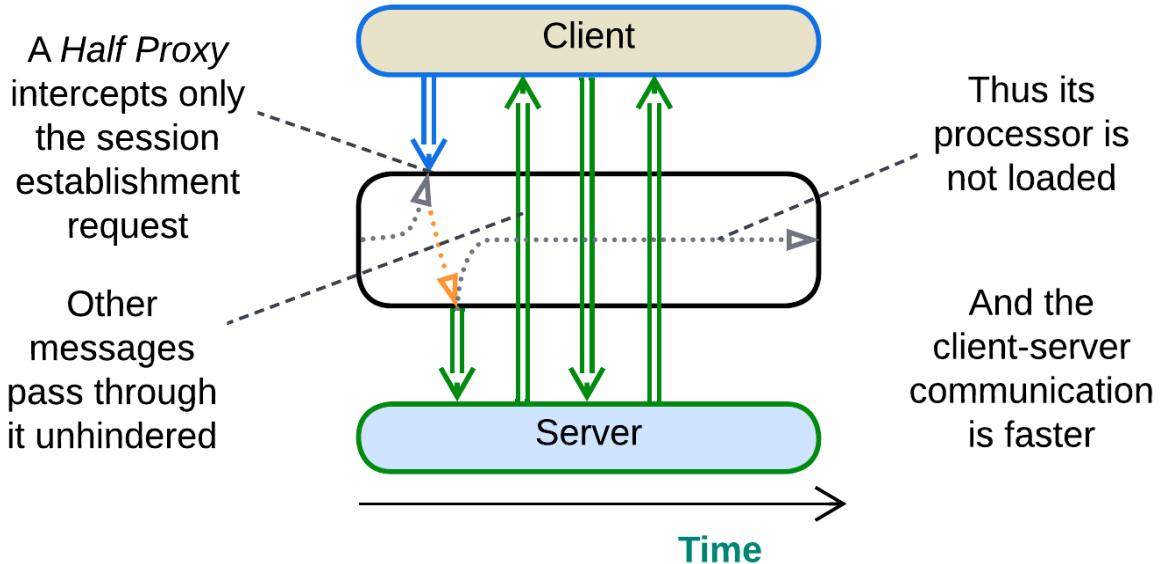
A *Proxy* [may either fully isolate the system it represents or merely help establish connections](#) between clients and servers. This resembles [closed and open layers](#) because a *Proxy* is a [layer](#) between a system and its clients.

Full Proxy



A *Full Proxy* processes every message between the system and its clients. It completely isolates the system and may meddle with the protocols but it is resource-heavy and adds to latency. [Adapters](#) and [Response Caches](#) are always *Full Proxies*.

Half-Proxy

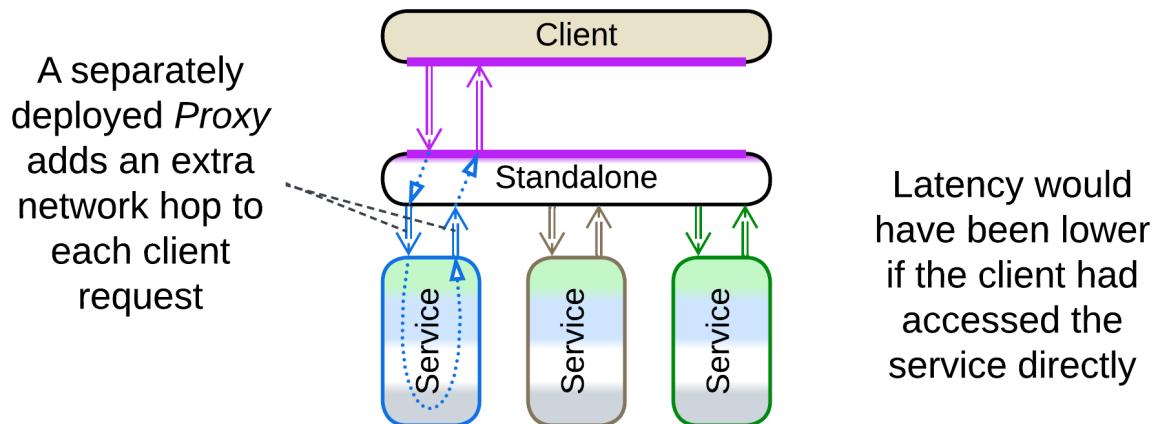


A *Half-Proxy* intercepts, analyzes, and routes the session establishment request from a client but then goes out of the loop. It may still forward the subsequent messages without looking into their content or it may even help connect the client and server directly, which is known as [direct server return](#) (DSR). This approach is faster and much less resource-hungry but is also less secure and less flexible than that of *Full Proxy*. A [Firewall](#), [Load Balancer](#), or [Reverse Proxy](#) may act as a *Half-Proxy*. IP telephony servers often use DSR: the server helps call parties find each other and establish direct media communication.

Variants by placement

As a *Proxy* stands between a (sub)system and its client(s), we can imagine a few ways to deploy it:

Separate deployment: Standalone

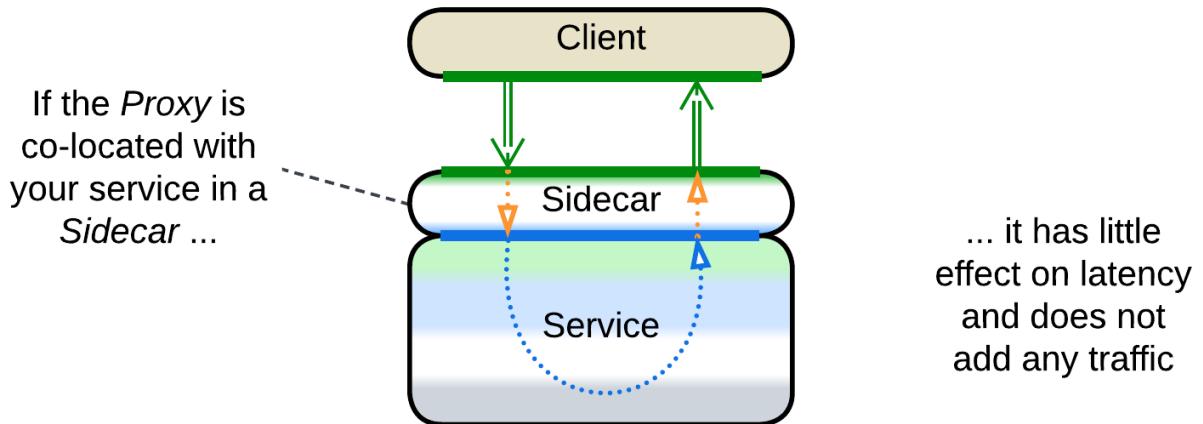


We can deploy a *Proxy* as a separate system component. This has the downside of an extra network hop (higher latency) in the way of every client's request to the system and back but that is unavoidable in the following cases:

- The *Proxy* uses a lot of system resources, thus it cannot be colocated with another component. This mostly affects [Firewall](#) and [Cache](#).

- The *Proxy* is stateful and deals with multiple services, which is true for a [Load Balancer](#), [Reverse Proxy](#), or [API Gateway](#).

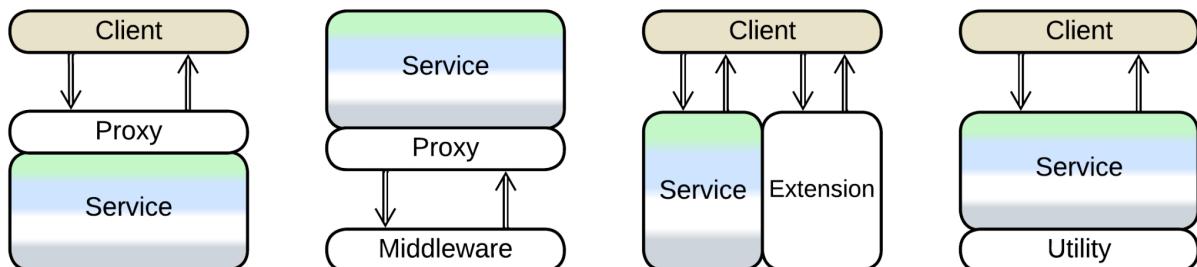
On the system side: Sidecar



We can often co-locate a *Proxy* with our system when the latter is not distributed. That avoids the extra network delay, traffic, operational complexity and does not add any new hardware which can fail at the most untimely moments. Such a placement is called *Sidecar* [DDS] and it is mostly applicable to [Adapters](#).

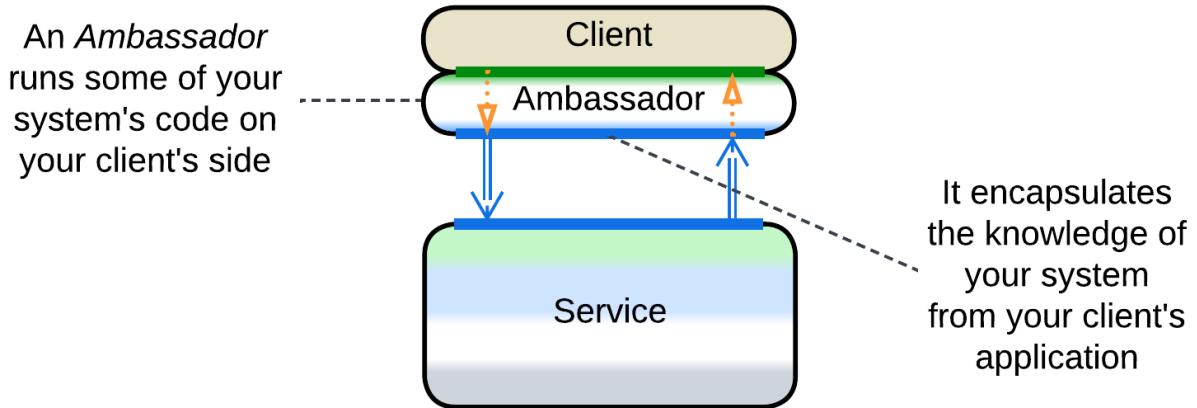
It should be noted that *Sidecar* – co-locating a generic component and business logic – is more of a DevOps approach than an architectural pattern, thus we can see it used in a variety of ways [DDS]:

- As a *Proxy* between a component and its clients.
- As an extra [service](#) that provides observability or configures the main service.
- As a [layer](#) with general-purpose utilities.
- As an [Adapter](#) for [Middleware](#).



[Service Mesh](#) ([Middleware](#) for [Microservices](#)) makes heavy use of *Sidecars*.

On the client side: Ambassador



Finally, a *Proxy* may be co-located with a component's clients, making it an *Ambassador* [DDS]. Its use cases include:

- Low-latency systems with *stateful shards* – each client should access the shard that has their data, which the *Proxy* knows how to choose.
- *Adapters* that help client applications use an optimized or secure protocol.

Variants by function

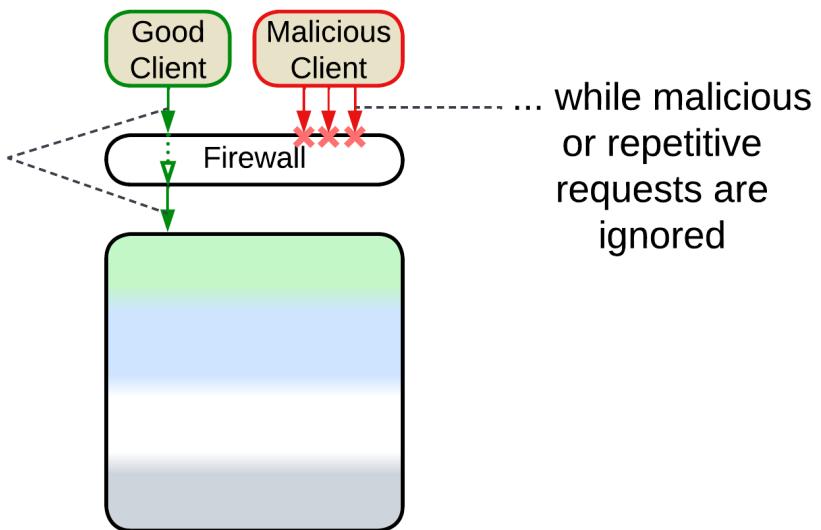
Proxies are ubiquitous in backend systems as using one or several of them frees the underlying code from the need to provide boilerplate non-business-logic functionality. It is common to have several kinds of *Proxies* deployed sequentially (e.g. *API Gateways* behind *Load Balancers* behind a *Firewall*) with many of them *pooled* to improve performance and stability. It is also possible to employ multiple kinds of *Proxies*, each serving its own kind of client, in parallel, resulting in *Backends for Frontends*.

As *Proxies* are used for many purposes, there are a variety of their specializations and names. Below is a very rough categorization, complicated by the fact that real-world *Proxies* often implement several categories at once.

For example, NGINX claims to be: an HTTP web server, *Reverse Proxy*, content *Cache*, *Load Balancer*, TCP/UDP *Proxy* server, and mail *Proxy* server – all at once.

Firewall, (API) Rate Limiter, API Throttling

A *Firewall* forwards valid requests to the system which it protects ...

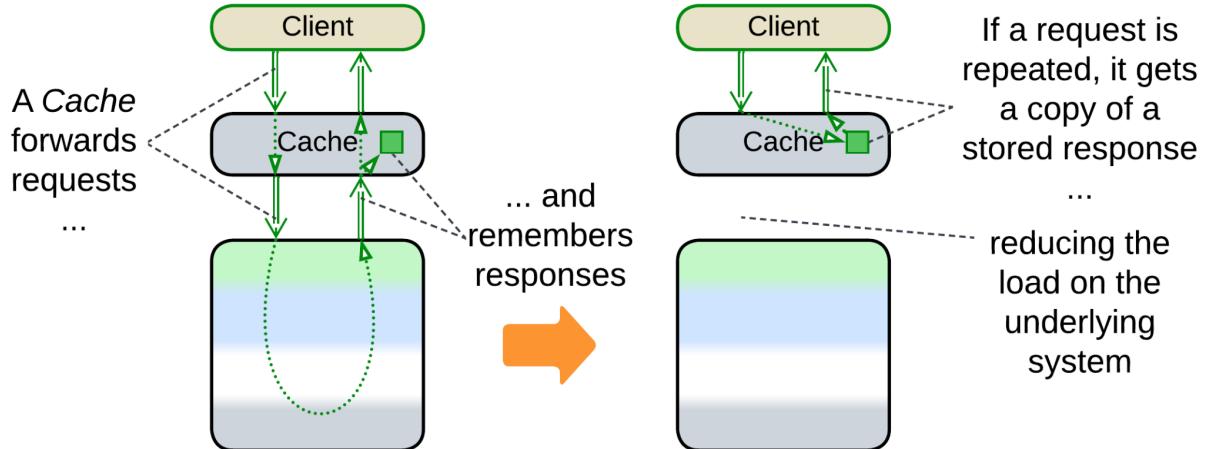


The *Firewall* is a component for white- and black-listing network traffic, mostly to protect against attacks. It is possible to use both generic hardware *Firewalls* on the external perimeter for brute force (D)DoS protection and more complex access rules at a second layer of software *Firewalls* to protect critical data and services from unauthorized access.

[*Rate Limiting*](#) makes sure that no single client uses too much of the system's resources – it sets a limit on how many requests from a single source the system can process over a unit of time. Any requests over the limit are rejected.

Throttling differs from *Rate Limiting* in that over-the-limit requests are queued for later processing, effectively slowing down communication with aggressive clients.

Response Cache, Read-Through Cache, Write-Through Cache, Write-Behind Cache, Cache, Caching Layer, Distributed Cache, Replicated Cache



If a system often gets identical requests, it is possible to remember its responses to most frequent of them and return the cached response without fully re-processing the request. The real thing is more complicated because users tend to change the data which the system stores, necessitating a variety of *cache refresh policies*. A *Response Cache* may be co-located with a *Load Balancer* or it may be [\[DDS\] sharded](#) (each Cache processes a unique

subset of requests) and/or [replicated](#) (all the Caches are similar) and thus require a *Load Balancer* of its own.

It is called *Response Cache* because it stores the system's responses to requests of its users or just *Cache* [DDS] because it is the most common kind of *Cache* in system architecture.

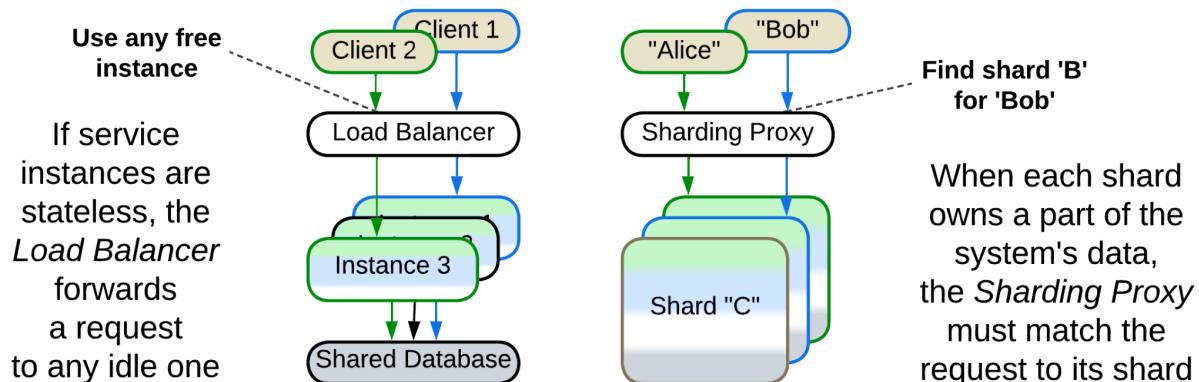
If the cached subsystem is a database, we can discern between read and write requests:

- [Read-Through Cache](#) is when the *Cache* is updated on a miss for a read request but is transparent to or invalidated by write requests.
- [Write-Through Cache](#) is when the *Cache* is updated by write requests that pass through it.
- [Write-Behind](#) is when the *Cache* aggregates multiple write requests to later send them to the database as a batch, saving bandwidth and possibly merging multiple updates of the same key.

It is possible to combine multiple servers into a virtual *Caching Layer* [DDS]:

- In the simplest case, which does not require any additional instrumentation aside from a [Load Balancer](#), the instances of the *Caches* are independent and may return stale results.
- In a *Distributed Cache*, driven by a [Sharding Proxy](#), every server (*shard*) holds a subset of the cached data, thus allowing for caching datasets which don't fit in a single computer's memory.
- In a *Replicated Cache* the datasets of all the servers are identical and synchronized on any modification. This scales the cache's throughput but requires a kind of synchronization engine, like a [Data Grid](#).

Load Balancer, Sharding Proxy, Cell Router, Messaging Grid, Scheduler



Here we have a hardware or software component which distributes user traffic among multiple [instances](#) of a service:

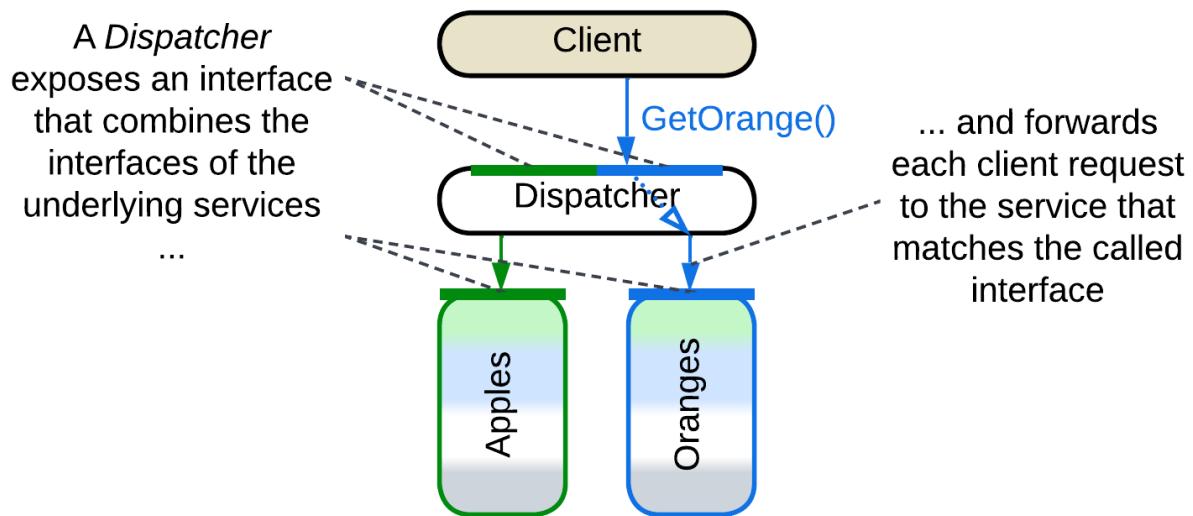
- A *Sharding Proxy* [DDS] selects a *shard* based on specific data which is present in a request (OSI level 7 request routing) for a system where each shard owns a part of the system's state, therefore only one (or a few for [replicated](#) shards) of the shards has the data required to process the client's request.
- A [Load Balancer](#) [DDS] for a [Pool of stateless instances](#) or [Replicas](#), or a *Messaging Grid* [FSA] of [Space-Based Architecture](#) evenly distributes the incoming traffic over identical request processors (OSI level 4 load balancing) to protect any instance of the underlying system from overload. In some cases it needs to be session-aware

(process OSI level 7) to assure that all the requests from a client are forwarded to the same instance of the service [DDS].

- It may forward read requests to [Read-Only Replicas](#) of the data while write requests are sent to the *master* database ([CQRS](#)-like behavior).
- A [Cell Router](#) chooses a data center which is the closest to the user's location.

Load Balancers are very common in high-load backends. High-availability systems deploy multiple instances of a *Load Balancer* in parallel to remain functional if one of the *Load Balancers* fails. CPU-intensive applications (like 3D games) often post asynchronous tasks for execution by *Thread Pools* under the supervision of a *Scheduler*. A similar pattern is found in OS kernels and *fiber* or *actor* frameworks where a limited set of CPU-affined threads is scheduled to run a much larger number of tasks.

Dispatcher, Reverse Proxy, Ingress Controller, Edge Service, Microgateway

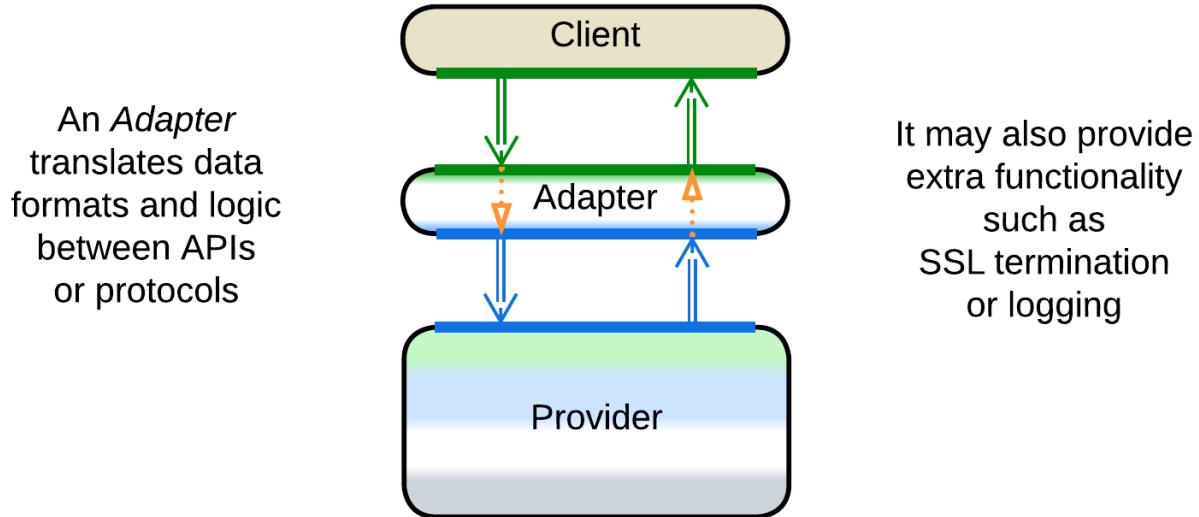


The [Reverse Proxy](#), [Ingress Controller](#), [Edge Service](#), or [Microgateway](#) is a router that stands between the Internet and the organization's internal network. It allows clients to use a public address for the system without knowing how and where their requests are processed. It parses user requests and forwards them to an internal server based on the requests' body. A *Reverse Proxy* can be extended with a *firewall*, *SSL termination*, *load balancing*, and *caching* functionality. Examples include Nginx.

Dispatcher [[POSA1](#)] is a similar component for a single-process application. It serves a complex command line interface by receiving and preprocessing user commands only to forward each command to a module which knows how to handle it. The modules may register their commands with the *Dispatcher* at startup or there may be a static dispatch table in the code.

You could have noticed that *Dispatcher* or *Reverse Proxy* is quite similar to *Load Balancer* or *Sharding Proxy* – they differ mostly in what kind of system lies below them: [Services](#) or [Shards](#).

Adapter, Anticorruption Layer, Open Host Service, Gateway, Message Translator, API Service, Cell Gateway, (inexact) Backend for Frontend, Hardware Abstraction Layer (HAL), Operating System Abstraction Layer (OSAL), Platform Abstraction Layer (PAL), Database Abstraction Layer (DBAL or DAL), Database Access Layer, Data Mapper, Repository



An *Adapter* [[GoF](#), [DDS](#)] is a mostly stateless *Proxy* that translates between an internal and public protocol and API formats. It may often be co-located with a *Reverse Proxy*. When it adapts messages, it is called a *Message Translator* [[EIP](#), [POSA4](#)].

As an *Adapter* adapts in two directions, it is often found between two components (in [Hexagonal Architecture](#)) or between a component and [Middleware](#) (in [Enterprise Service Bus](#) and [Service Mesh](#)).

In [[DDD](#)], when one component (*consumer*) depends on another (*supplier*), there may be an *Adapter* in between to decouple them. It is called *Anticorruption Layer* [[DDD](#)] when owned by the *consumer's* team or *Open Host Service* [[DDD](#)] if the *supplier* adds it to grant one or more stable interfaces (*Published Languages* [[DDD](#)]).

A *Gateway* [[PEAA](#)] or [API Service](#) often implies an *Adapter* with extra functionality, like *Reverse Proxy*, authorization and authentication. [Cell Gateway](#) is a *Gateway* for a [Cell](#).

When a *Gateway* translates a single public API method into several calls towards internal services, it becomes an *API Gateway* [[MP](#)] which is an aggregate of *Proxy* (for protocol translation) and [Orchestrator](#).

An *Adapter* between an end-user client (web interface, mobile application, etc.) and the system's API is often called [Backend for Frontend](#). It decouples the UI from the backend-owned system's API, giving the teams behind both components freedom to work with less synchronization.

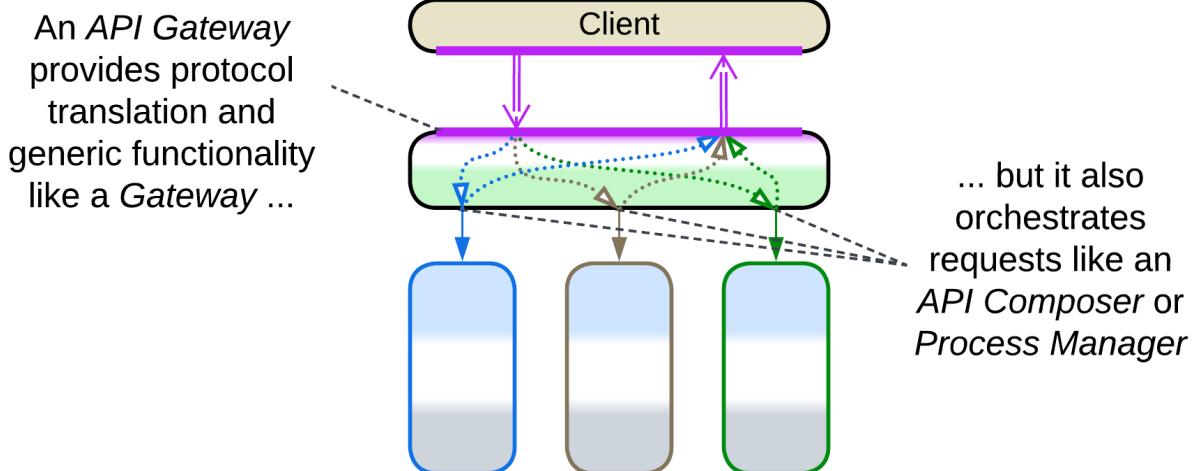
There is also a whole bunch of *Adapters* that aim to protect the business logic from its environment, the idea which is perfected by [Hexagonal Architecture](#):

- [Hardware Abstraction Layer](#) (HAL) hides details of hardware to make the code portable.
- [Operating System Abstraction Layer](#) (OSAL) or [Platform Abstraction Layer](#) (PAL) abstracts the OS to make the application cross-platform.

- [Database Abstraction Layer](#) (*DBAL* or *DAL*), [Database Access Layer](#) [[POSA4](#)] or [Data Mapper](#) [[PEAA](#)] attempts to help building database-agnostic applications by making all the databases look the same.
- [Repository](#) [[PEAA](#), [DDD](#)] provides methods to access a record stored in a database as if it were an object in the application's memory.

An *Adapter* creates a [layer of indirection](#) between your code and a library or service which it uses. If the external component's interface changes, or you need to substitute the thing with an incompatible implementation from another vendor, and your code accesses the component directly, you will have to make many changes throughout your code. However, if there is an *Adapter* in-between, your code depends only on the interface of the *Adapter*. And when the external component changes or is replaced, only the relatively small *Adapter*'s implementation needs to change while your main code is blessed with ignorance of what lies beyond the *Adapter*'s borders.

API Gateway



API Gateway [[MP](#)] is a fusion of [Gateway](#) (*Proxy*) and [API Composer](#) (*Orchestrator*). The *Gateway* aspect encapsulates the external (public) protocol while the *API Composer* translates the system's high-level public API methods into multiple (usually parallel) calls to the APIs of internal components, collects the results and conjoins them into a response.

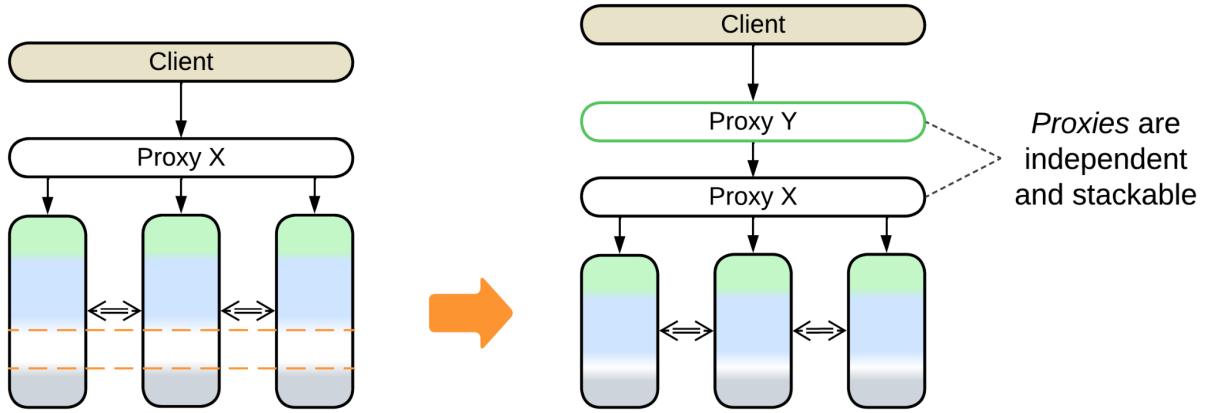
API Gateway is [discussed in more detail](#) under *Orchestrator*.

Evolutions

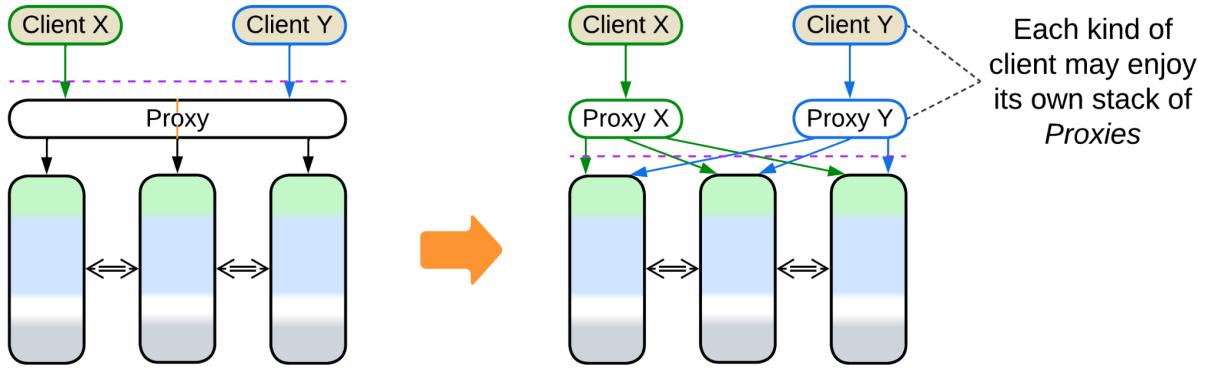
It usually makes little sense to get rid of a *Proxy* once it has been integrated into a system. The only real drawback to using a *Proxy* is a slight increase in latency for user requests which may be mitigated through the creation of [bypass channels](#) between the clients and a service that needs low latency. The other drawback of the pattern, the *Proxy* being a single point of failure, is countered by deploying multiple instances of the *Proxy*.

As *Proxies* are usually third-party products, there is not much we can change about them:

- We can add another kind of a *Proxy* on top of an existing one.



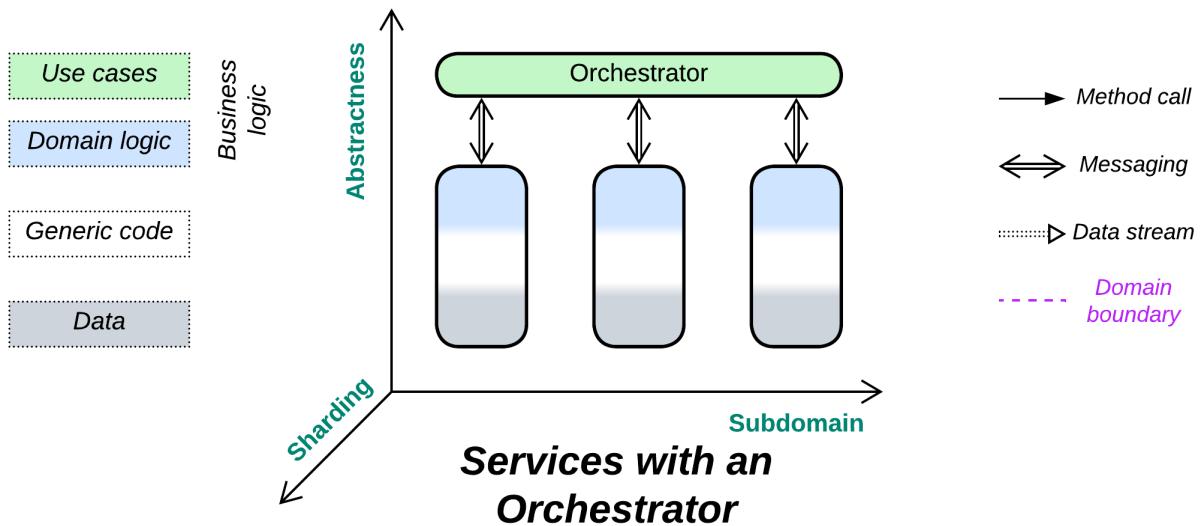
- We can use a stack of *Proxies* per client, making *Backends for Frontends*.



Summary

A *Proxy* represents your system to its clients and takes care of some aspects of the communication. It is common to see multiple *Proxies* deployed sequentially as they are often stackable.

Orchestrator



One ring to rule them all. Make a service to integrate other services.

Known as: Orchestrator [[MP](#), [FSA](#)], Orchestrated Services, Service Layer [[PEAA](#)], Application Layer [[DDD](#)], Wrapper Facade [[POSA4](#)], Multi-Worker [[DDS](#)], Controller / Control, Workflow Owner [[FSA](#)] of [Microservices](#), and Processing Grid [[FSA](#)] of [Space-Based Architecture](#).

Aspects:

- Mediator [[GoF](#), [SAHP](#)],
- Facade [[GoF](#)].

Variants:

By transparency:

- Closed or strict,
- Open or relaxed.

By structure (not exclusive):

- Monolithic,
- Sharded,
- Layered [[FSA](#)],
- A service per client type ([Backends for Frontends](#)),
- A service per subdomain [[FSA](#)] ([Hierarchy](#)),
- A service per use case [[SAHP](#)] ([SOA](#)-style).

By function:

- API Composer [[MP](#)] / Remote Facade [[PEAA](#)] / [Gateway Aggregation](#) / Composed Message Processor [[EIP](#)] / Scatter-Gather [[EIP](#), [DDS](#)] / MapReduce [[DDS](#)],
- Process Manager [[EIP](#), [LDDD](#)] / Orchestrator [[FSA](#)],
- (Orchestrated) Saga [[LDDD](#)] / Saga Orchestrator [[MP](#)] / [Saga Execution Component](#) / Transaction Script [[PEAA](#), [LDDD](#)] / Coordinator [[POSA3](#)],
- [Integration \(Micro-\)Service](#) / Application Service,
- (with a [Gateway](#)) API Gateway [[MP](#)] / Microgateway,
- (with a [Middleware](#)) Event Mediator [[FSA](#)],
- (with a [Middleware](#) and [Adapters](#)) Enterprise Service Bus (ESB) [[FSA](#)].

Structure: A layer of high-level business logic built on top of lower-level services.

Type: Extension.

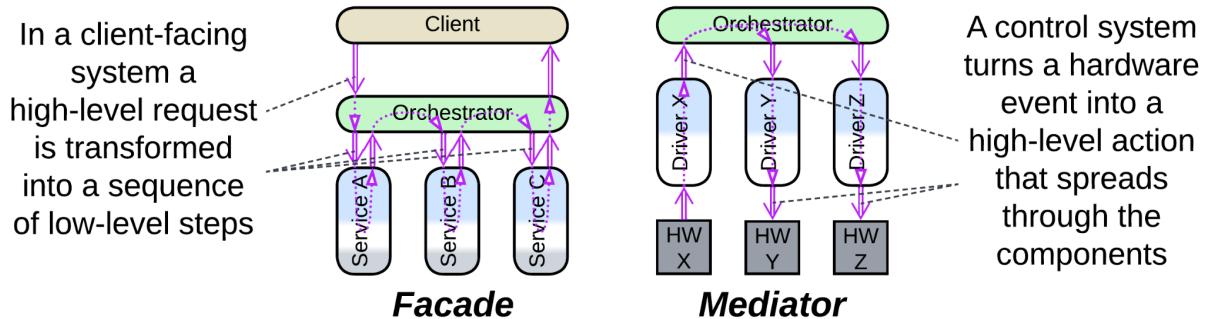
Benefits	Drawbacks
Separates integration concerns from the services – decouples the services' APIs	May increase latency for global use cases
Global use cases can be changed and deployed independently from the services	Qualities of the services become coupled to an extent
Decouples the services from the system's clients	API design is an extra step before implementation

References: [FSA] discusses orchestration in its chapters on *Event-Driven Architecture*, *Service-Oriented Architecture*, and *Microservices*. [MP] describes orchestration-based Sagas and its Order Service acts as an *Application Service* without explicitly naming the pattern. [POSA4] defines several variants of *Facade*.

An *Orchestrator* takes care of global use cases (those involving multiple services) thus allowing each service to specialize in its own subdomain and, ideally, forget about the existence of all the other services. This way the entire system's high-level logic (which is subject to frequent changes) is kept (and deployed) together, isolated from usually more complex subdomain-specific services. Dedicating a *layer* to global scenarios makes them relatively easy to implement and debug, while the corresponding development team that communicates with clients shelters other narrow-focused teams from disruptions. The cost of employing an *Orchestrator* is both degraded performance when compared to basic *Services* that rely on *choreography* [FSA, MP] and some coupling of the properties of the orchestrated services as the *Orchestrator* usually treats every service in the same way.

An *Orchestrator* fulfills two closely related roles:

- As a *Mediator* [GoF, SAHP] it keeps the states of the underlying components (services) consistent by propagating changes that originate in one component to the rest of the system. This role is prominent in *control software*, pervading automotive, aerospace, and IoT industries. The *Mediator* role also emerges as *Saga* [MP].
- As a *Facade* [GoF] it builds high-level scenarios out of smaller steps provided by the services or modules it controls. This role is obvious for *processing systems* where clients communicate with the *Facade*, but it is also featured in *control* software, because sometimes a simple event may trigger a complex multi-component scenario managed by the system's *Orchestrator*.

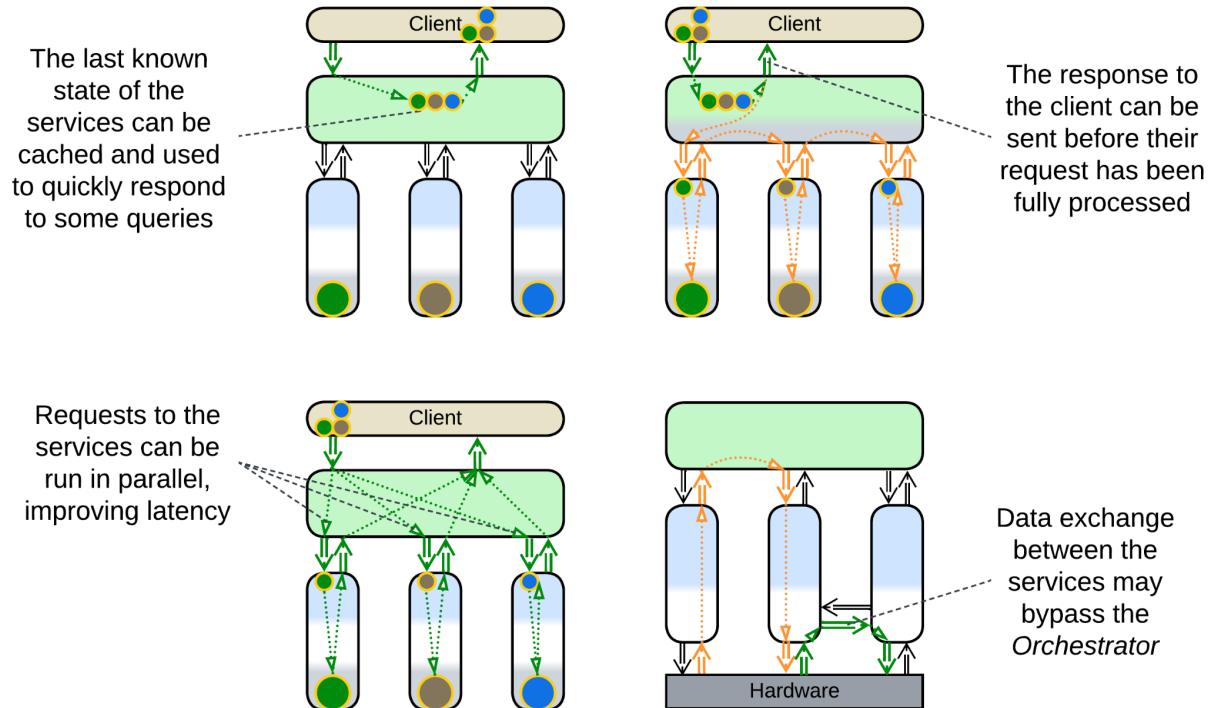


Data processing systems, such as backends, may deploy multiple *instances* of stateless *Orchestrators* to improve stability and performance. In contrast, an *Orchestrator* in *control software* incorporates the highest-level view of the system's state thus it cannot be easily replicated (as any replicated state must be kept synchronized, introducing delay or inconsistency in decision-making).

Performance

When compared to [choreography](#), [orchestration](#) usually worsens latency as it involves extra steps of communication between the *Orchestrator* and orchestrated components. However, the effects should be estimated on case by case basis, as there are exceptions in at least the following cases:

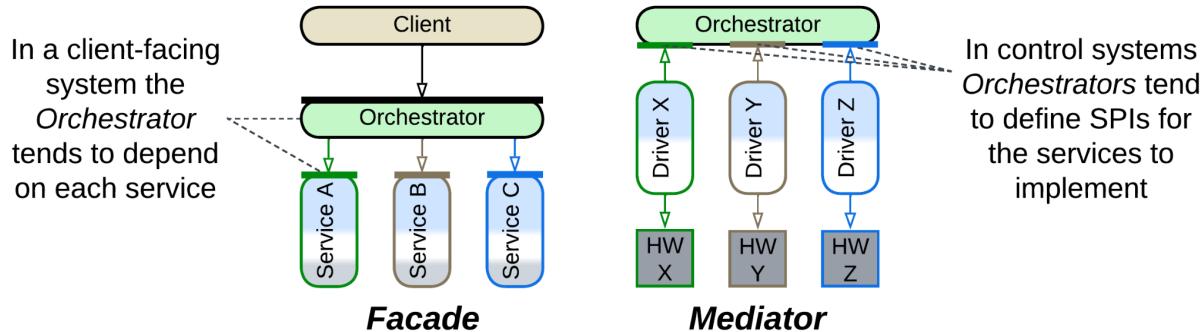
- An *Orchestrator* may cache the state of the orchestrated system, gaining the ability to immediately respond to read requests with no need to query the underlying components. This is very common with [control systems](#).
- An *Orchestrator* may persist a write request, respond to the client, and then start the actual processing. Persistence grants that the request will eventually be completed as it can be restarted.
- An *Orchestrator* may run multiple subrequests in parallel, reducing latency compared to a chain of choreographed events.
- In a highly loaded or latency-critical system, orchestrated services may establish direct data streams that bypass the *Orchestrator*. A classic example is [VoIP](#) where the call establishment logic (SIP) goes through an orchestrating server while the voice or video (RTP) is streaming directly between the clients.



I don't see how orchestration can affect throughput as in most cases the *Orchestrator* can be scaled. However, scaling weakens consistency as then no instance of the *Orchestrator* has exclusive control over the system's state.

Dependencies

An *Orchestrator* may depend on the *APIs* of the services it orchestrates or define *SPIs* for them to implement, with the first mode being natural for its *Facade* [GoF] aspect and the second one for the *Mediator* [GoF]:



If an *Orchestrator* is added to integrate existing components, it will use their APIs.

In large projects, where each service gets a separate team, the APIs need to be negotiated beforehand, and will likely be owned by the orchestrated services.

Smaller (single-team) systems tend to be developed top-down, with the *Orchestrator* being the first component to implement, thus it defines the interfaces it uses.

Likewise, [control systems](#) tend to reverse the dependencies, with their services depending on the orchestrator's SPI – either because their events originate with the services (so the services must have an easy way to contact the *Orchestrator*) or to provide for polymorphism between the low-level components. See the [chapter on orchestration](#) for more details.

Applicability

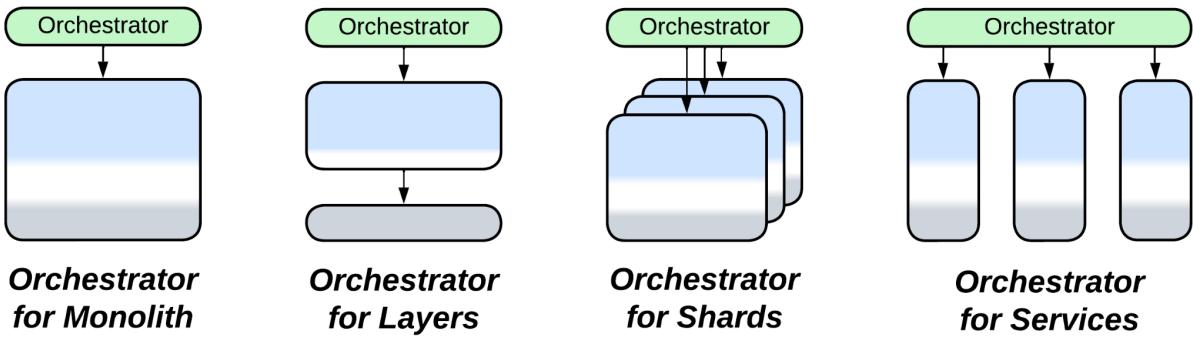
Orchestrators shine with:

- *Large projects*. The partition of business logic into a high-level application (*Orchestrator*) and the multiple [subdomain Services](#) it relies on provides perfect code decoupling and team specialization.
- *Specialized teams*. As an improvement over [Services](#), the teams which develop deep knowledge of subdomains will delegate communication with customers to the application team.
- *Complex and unstable requirements*. The *integration layer* (*Orchestrator*) should be high-level and simple enough to be easily extended or modified to cover most of the customer requests or marketing experiments without much help from the domain teams.

Orchestrators fail in:

- *Huge projects*. At least one aspect of complexity is going to hurt. Either the number of the subdomain services and the size of their APIs will make it impossible for an *Orchestrator* programmer to find the correct methods to call, or the *Orchestrator* itself will become unmanageable due to the number and length of its use cases. This can be addressed by dividing the *Orchestrator* into a layer of services (resulting in [Backends for Frontends](#) or [Cell-Based Architecture](#)) or multiple layers (often yielding [Top-Down Hierarchy](#)). It is also possible to go for the [Service-Oriented Architecture](#) as that has more fine-grained components.
- *Small projects*. The implementation overhead of defining and stabilizing service APIs and the performance penalty of the extra network hop may outweigh the extra flexibility of having the *Orchestrator* as a separate system component.
- *Low latency*. Any system-wide use case will make multiple calls between the application (*Orchestrator*) and services, with each interaction adding to the latency.

Relations



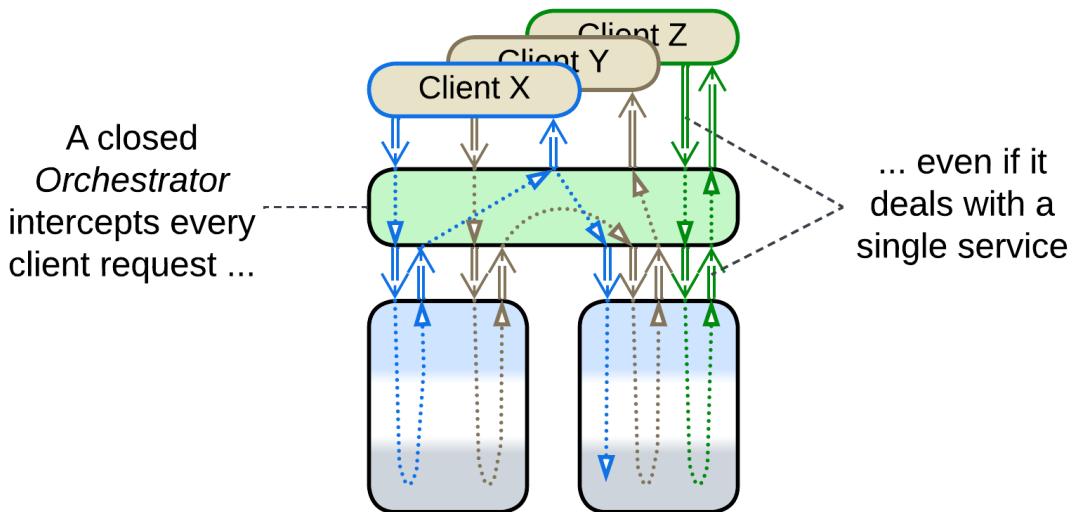
Orchestrator:

- Extends [Services](#) or, rarely, [Monolith](#), [Shards](#), or [Layers](#) (forming [Layers](#)).
- Can be merged with a [Proxy](#) into an [API Gateway](#), with a [Middleware](#) into an [Event Mediator](#), or with a [Middleware](#) and [Adapters](#) into an [Enterprise Service Bus](#).
- Is a special case (single service) of [Backends for Frontends](#), [Service-Oriented Architecture](#) or (2-layer) [Hierarchy](#).
- Can be implemented by a [Microkernel](#).

Variants by transparency

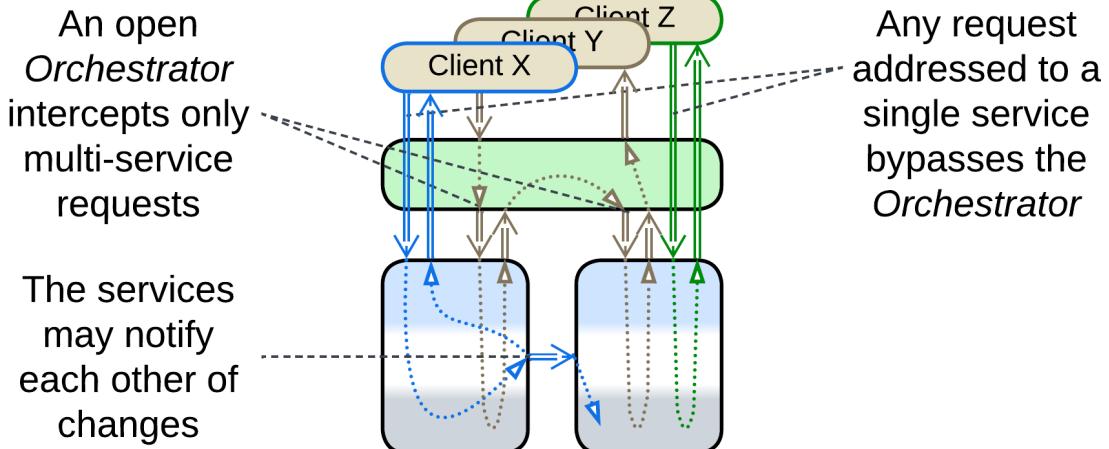
It seems that an *Orchestrator*, just like a *layer*, which it is, [can be open \(relaxed\)](#) or [closed \(strict\)](#):

Closed or strict



A *strict* or *closed Orchestrator* isolates the orchestrated services from their users – all the requests go through the *Orchestrator*, and the services don't need to intercommunicate.

Open or relaxed



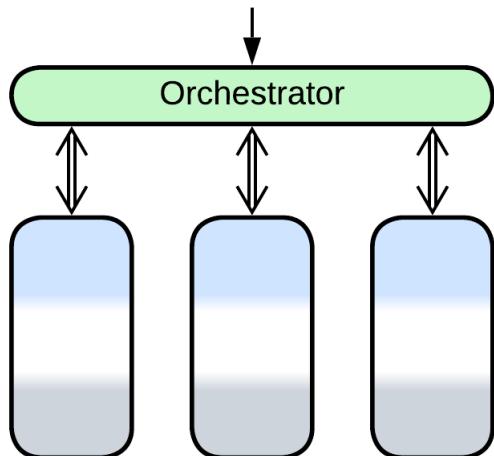
An *open Orchestrator* implements a subset of system-wide scenarios that require strict data consistency while less demanding requests go from the clients directly to the underlying services, which rely on *choreography* or *shared data* for communication. Such a system sacrifices the clarity of design to avoid some of the drawbacks of both *choreography* and *orchestration*:

- The orchestrator development team, which may be overloaded or slow to respond, is not involved in implementing the majority of use cases.
- Most of the use cases avoid the performance penalty caused by the orchestration.
- Failure of the *Orchestrator* does not disable the entire system.
- The relaxed *Orchestrator* still allows for synchronized changes of data in multiple services, which is rather hard to achieve with choreography.

Variants by structure (can be combined)

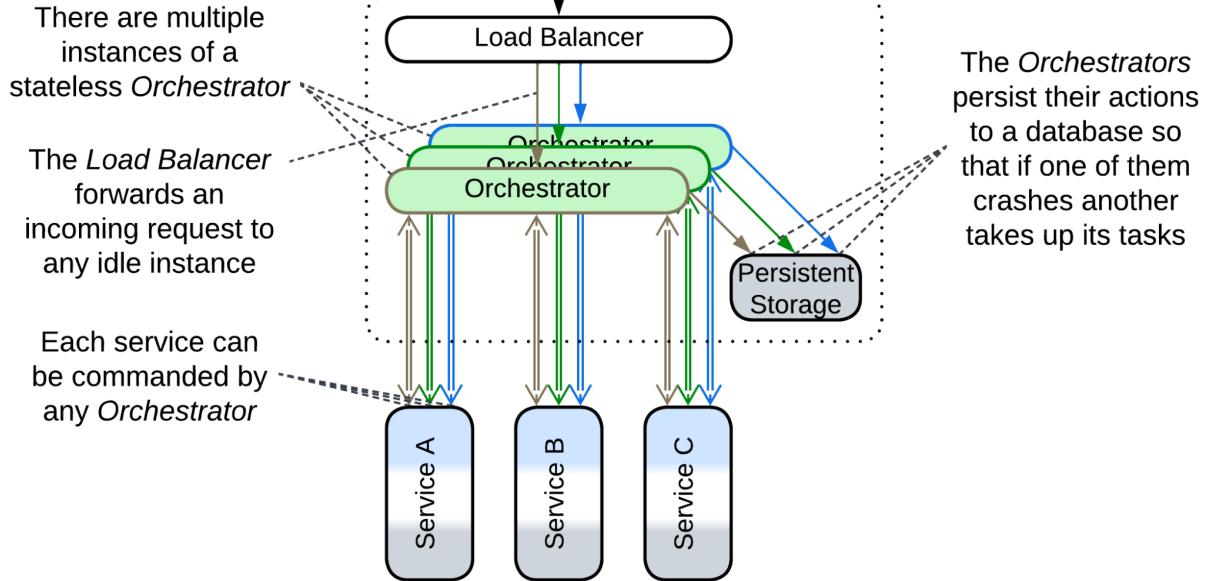
The orchestration (application [DDD] / [integration](#) / [composite](#)) layer has several structural (implementation) options:

Monolithic



A single *Orchestrator* is deployed. This option fits ordinary medium-sized projects.

Scaled

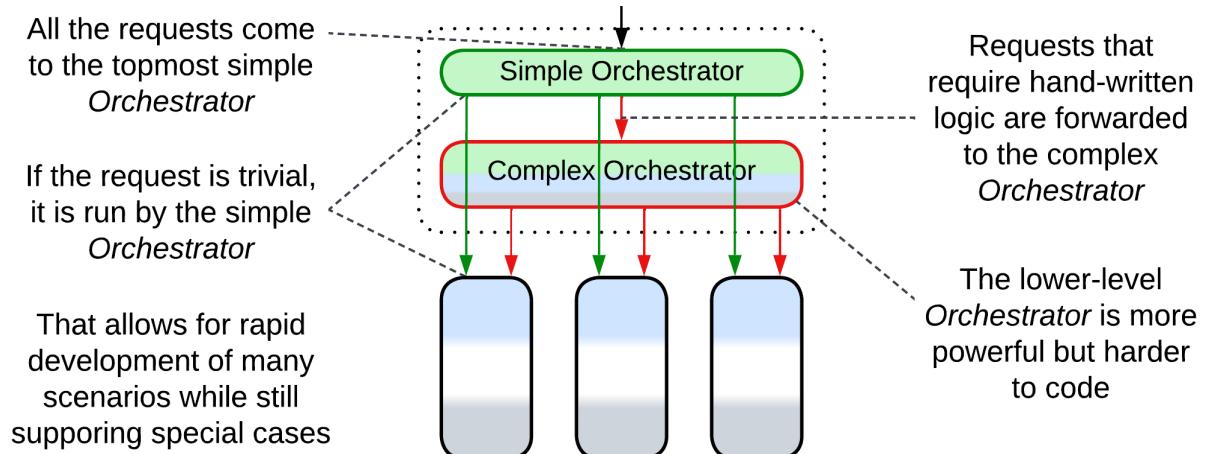


High availability requires multiple [instances](#) of a stateless *Orchestrator* to be deployed. A *Mediator* ([Saga](#), writing *Orchestrator*) may store the current transaction's state in a [Shared Database](#) to assure that if it crashes there is always another instance ready to take up its job.

High load systems also require multiple instances of *Orchestrators* because a single instance is not enough to handle the incoming traffic.

Not all data is made equal. For example, an online store has different requirements for reliability of its buyers' cart contents and the payments. If the current buyers' carts become empty when the store's server crashes, that makes only a minor annoyance. However, any financial data loss or a corrupted money transfer is a real trouble. Therefore an online store may implement its cart with a simple in-memory *Orchestrator*, but should be very careful about the payments workflow, every step of which must be persisted to a reliable database.

Layered



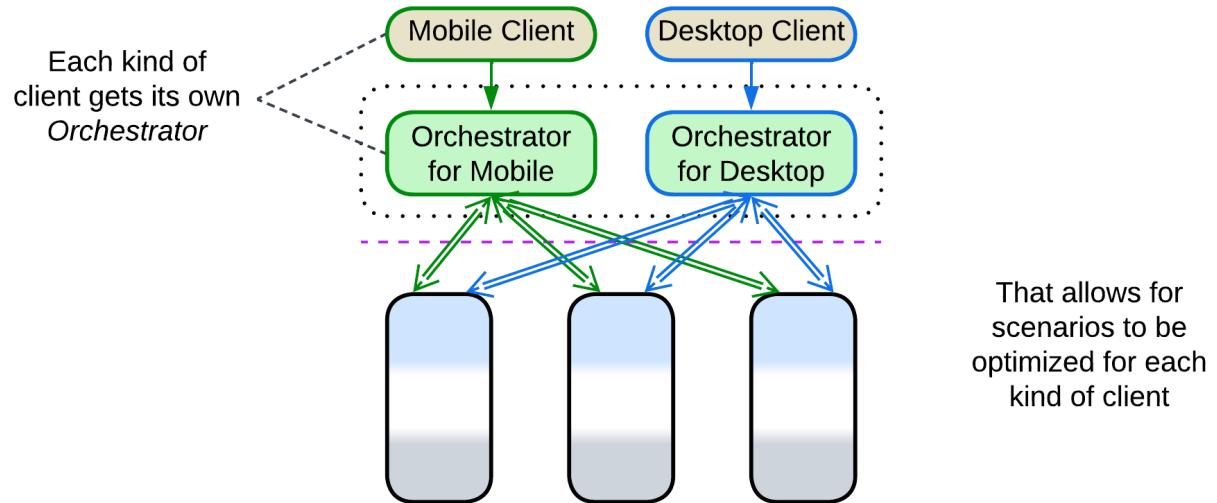
[[FSA](#)] describes an option of a layered [Event Mediator](#). A client's request comes to the topmost layer of the *Orchestrator* which uses the simplest (and least flexible) framework. If the request is found to be complex, it is forwarded to the second layer which is based on a

more powerful technology. And if it fails or requires a human decision then it is forwarded again to the even more complex custom-tailored orchestration layer.

That allows the developers to gain the benefits of a high-level declarative language in a vast majority of scenarios while falling back to hand-written code for a few complicated cases. The choice is not free as programmers need to learn multiple technologies, interlayer debugging is anything but easy and performance is likely to be worse than that of a monolithic *Orchestrator*.

A similar example is using an [API Composer](#) for the top layer, followed by a [Process Manager](#) and a [Saga Engine](#).

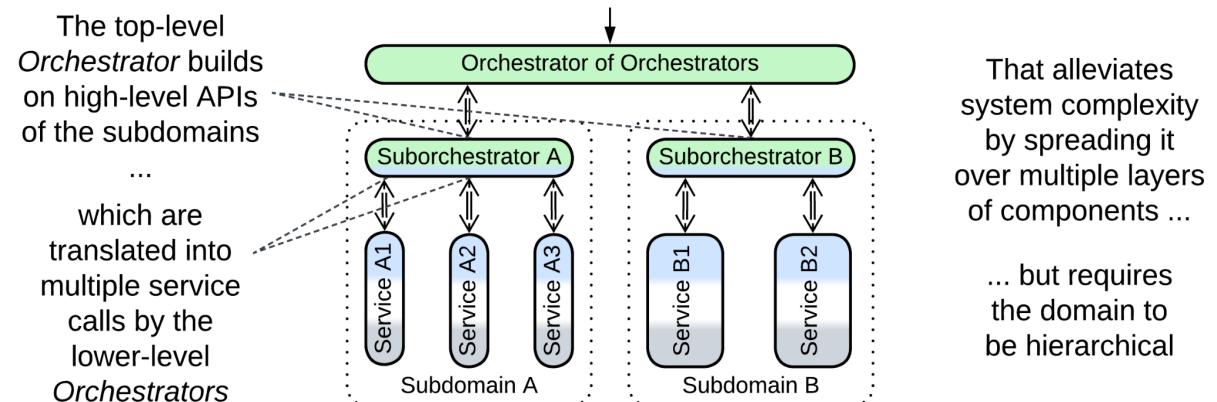
A service per client type (Backends for Frontends)



If your clients strongly differ in workflows (e.g. [OLAP](#) and [OLTP](#), or user and admin interfaces), implementing dedicated *Orchestrators* is an option to consider. That makes each client-specific *Orchestrator* smaller and more cohesive than the unified implementation would be and gives more independence to the teams responsible for different kinds of clients.

This pattern is known as [Backends for Frontends](#) and has a chapter of its own.

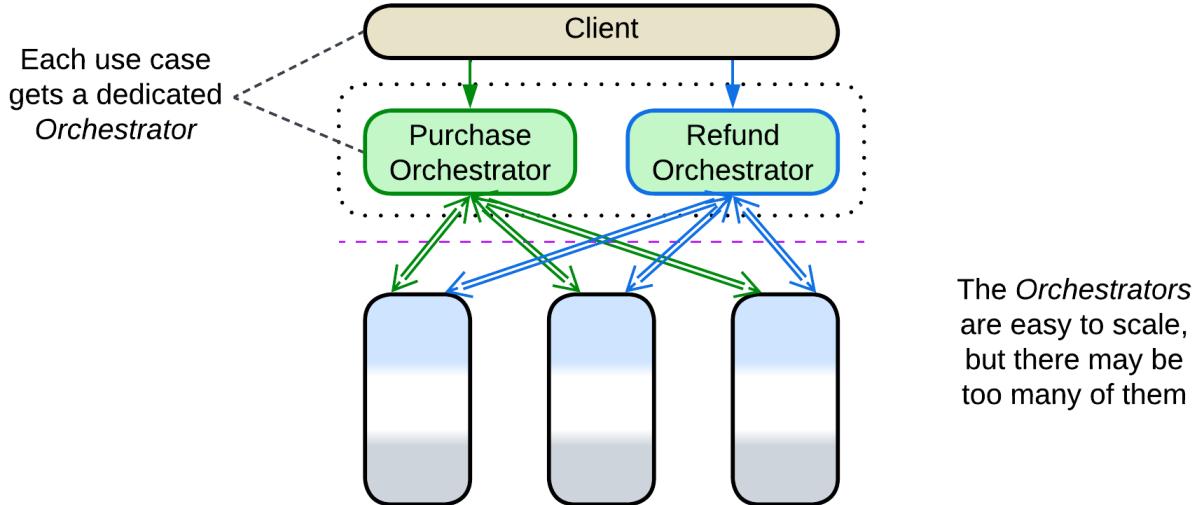
A service per subdomain (Hierarchy)



In large systems a single *Orchestrator* is very likely to become overgrown and turn into a development bottleneck (see [Enterprise Service Bus](#)). Building a [hierarchy](#) of *Orchestrators* may help [\[FSA\]](#), but only if the domain itself is hierarchical. The top-level component may even be a [Reverse Proxy](#) if no use cases cross subdomain borders or if the sub-

orchestrators employ [choreography](#), resulting in a flat [Cell-Based Architecture](#). Otherwise it is a tree-like [Orchestrator of Orchestrators](#).

A service per use case (SOA-style)

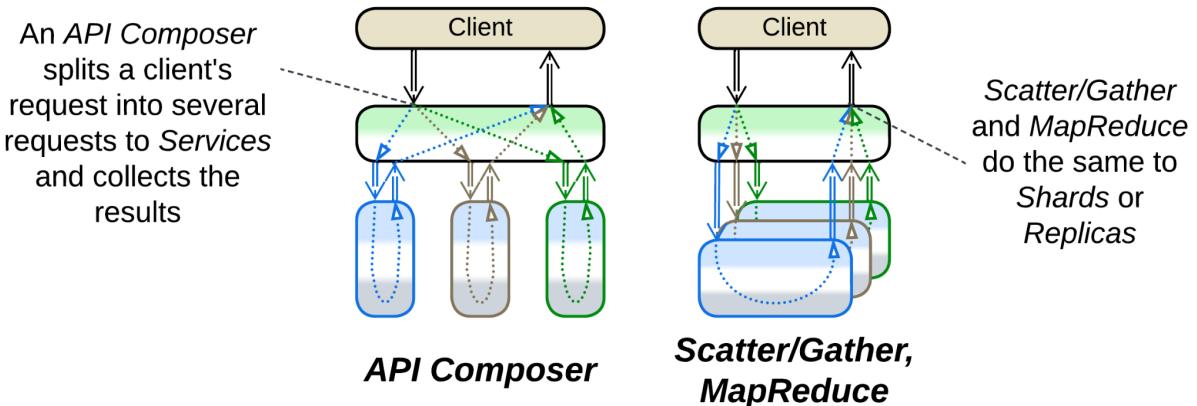


[SAHP] advises for single-purpose *Orchestrators* in [Microservices](#): each *Orchestrator* manages one use case. This enables fine-grained scalability but will quickly lead to integration hell as new scenarios keep getting added to the system. Overall, such a use of *Orchestrators* resembles the [task layer](#) of [SOA](#).

Variants by function

Orchestrators may function in slightly different ways:

API Composer, Remote Facade, Gateway Aggregation, Composed Message Processor, Scatter-Gather, MapReduce



API Composer [MP] is a kind of *Facade* [GoF] which decreases the system's latency by translating a high-level incoming message into a set of lower-level internal messages, sending them to the corresponding *services* in parallel, waiting for results and collecting the latter into a response to the original message. Such a logic may often be defined declaratively in a third-party tool without writing any low-level code. **Remote Facade** [PEAA] is a similar pattern which makes synchronous calls to the underlying components – it exists to implement a coarse-grained protocol with the system's clients, so that a client may

achieve whatever it needs through a single request. [Gateway Aggregation](#) is a generalization of these patterns.

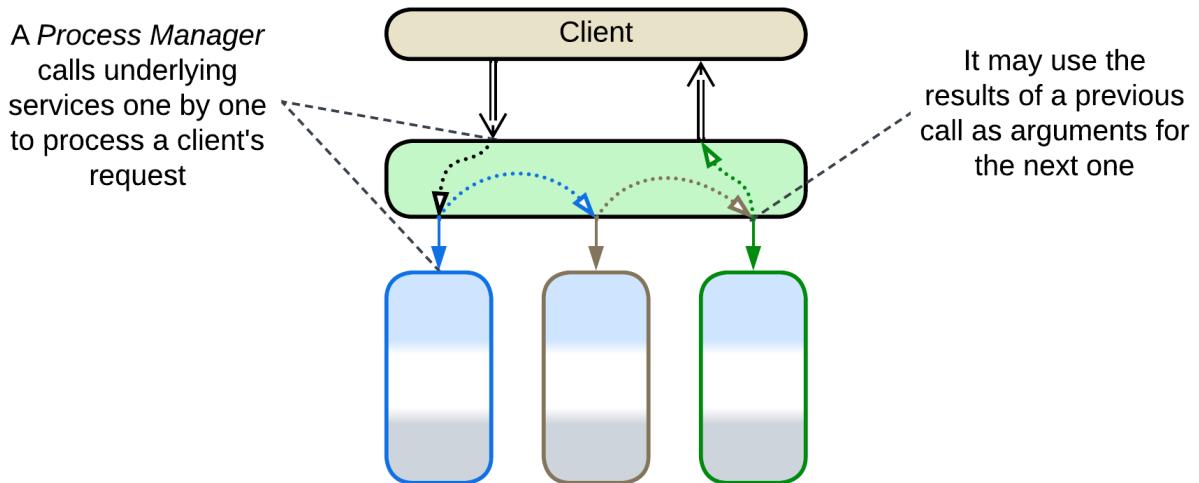
Composed Message Processor [EIP] disassembles *API Composer* into smaller components: it uses a *Splitter* [EIP] to subdivide the request into smaller parts, a *Router* [EIP] to send each part to its recipient, and an *Aggregator* [EIP] to collect the responses into a single message. Unlike *API Composer*, it can also address *Shards* or *Replicas*. A *Scatter-Gather* [EIP, DDS] broadcasts a copy of the incoming message to each recipient, thus it lacks a *Splitter* (though [DDS] seems to ignore this difference). *MapReduce* [DDS] is similar to *Scatter-Gather* except that it summarizes the results received in order to yield a single value instead of concatenating them.

If an *API Composer* needs to conduct sequential actions (e.g. first get user id by user name, then get user data by user id), it becomes a *Process Manager* which may require some coding.

An *API Composer* is usually deployed as a part of an [API Gateway](#).

Example: Microsoft has an [article](#) on aggregation.

Process Manager, Orchestrator



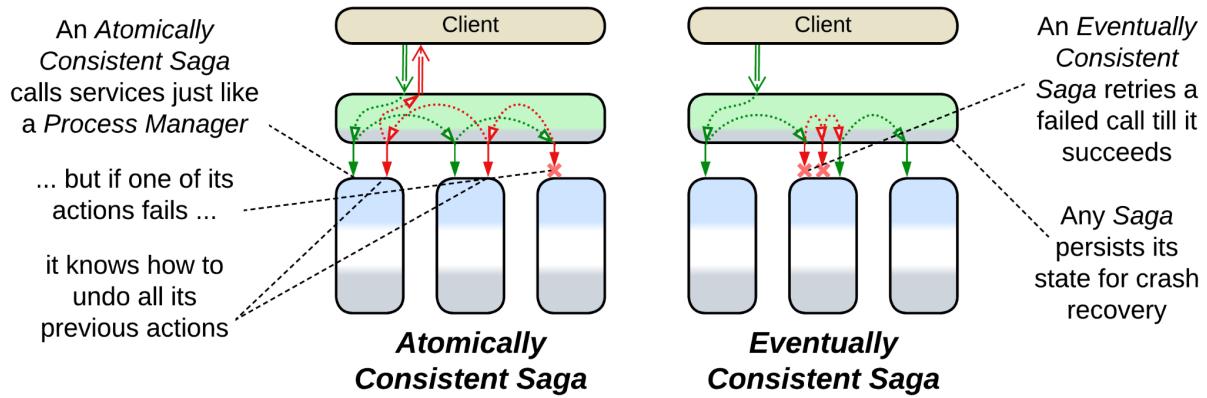
Process Manager [EIP, LDDD] (referred simply as *Orchestrator* in [FSA]) is a kind of *Facade* that translates high-level tasks into sequences of lower-level steps. This subtype of *Orchestrator* receives a client request, stores its state, runs pre-programmed request processing steps, and returns a response. Each of the steps of a *Process Manager* is similar to a whole task of an *API Composer* in that it generates a set of parallel requests to internal services, waits for the results and stores them for the future use in the following steps or final response. The scenarios it runs may branch on conditions.

A *Process Manager* may be implemented in a general-purpose programming language, a declarative description for a third-party tool, or a mixture thereof.

A *Process Manager* is usually a part of an [API Gateway](#), [Event Mediator](#) or [Enterprise Service Bus](#).

Example: [\[FSA\]](#) provides several examples.

(Orchestrated) Saga, Saga Orchestrator, Saga Execution Component, Transaction Script, Coordinator



(Orchestrated [[SAHP](#)]) [Saga](#) [[LDDD](#)], [Saga Orchestrator](#) [[MP](#)] or [Saga Execution Component](#) is a subtype of *Process Manager* which is specialized in *distributed transactions*.

An *Atomically Consistent Saga* [[SAHP](#)] (which is the default meaning of the term) comprises a pre-programmed sequence of {"do", "undo"} action pairs. When it is run, it iterates through the "do" sequence till it either completes (meaning that the transaction succeeded) or fails. A failed *Atomically Consistent Saga* begins iterating through its "undo" sequence to roll back the changes that were already made. In contrast, an *Eventually Consistent Saga* [[SAHP](#)] always retries its writes till all of them succeed.

A *Saga* is often programmed declaratively in a third-party [Saga Framework](#) which can be integrated into any service that needs to run a *distributed transaction*. However, it is quite likely that such a service itself is an [Integration Service](#) as it seems to [orchestrate](#) other services.

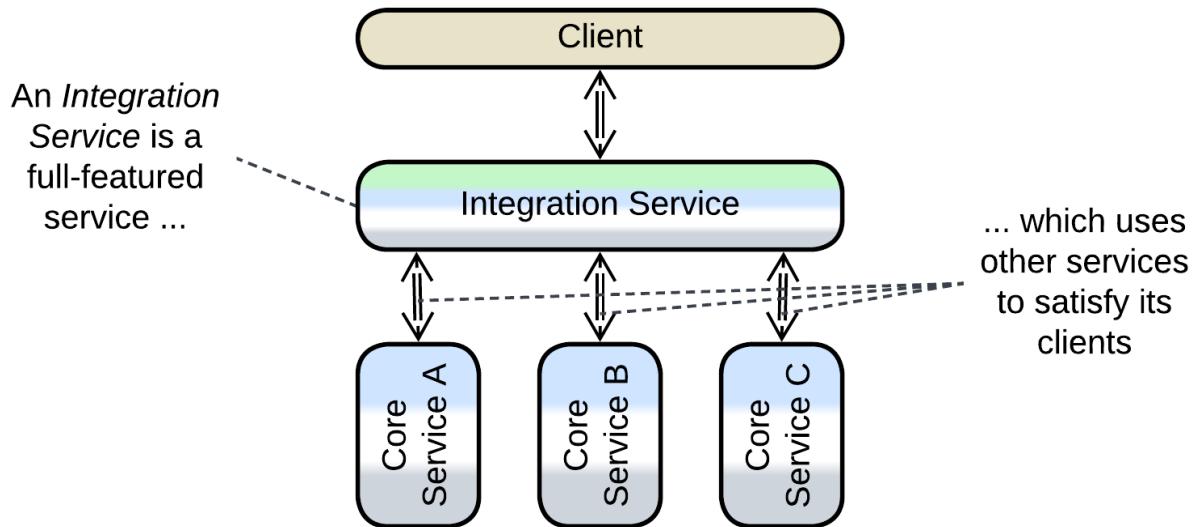
A *Saga* plays the roles of both *Facade* by translating a single transaction request into a series of calls to the services' APIs and *Mediator* by keeping the states of the services consistent (the transaction succeeds or fails as a whole). Sometimes a *Saga* may include requests to external services (which are not parts of the system you are developing).

A *Transaction Script* [[PEAA](#), [LDDD](#)] is a procedure that executes a transaction, possibly over multiple databases [[LDDD](#)]. Unlike a *Saga*, it is synchronous, written in a general programming language, and does not require a dedicated framework to run. It operates database(s) directly while a *Saga* usually sends commands to services. A *Transaction Script* may return data to its caller.

Coordinator [[POSA3](#)] is a generalized pattern for a component which manages multiple tasks (e.g. software updates of multiple components) to achieve "all or nothing" results (if any update fails, other components are rolled back).

Example: [[SAHP](#)] investigates many kinds of Sagas while [[MP](#)] has a shorter description.

Integration (Micro-)Service, Application Service



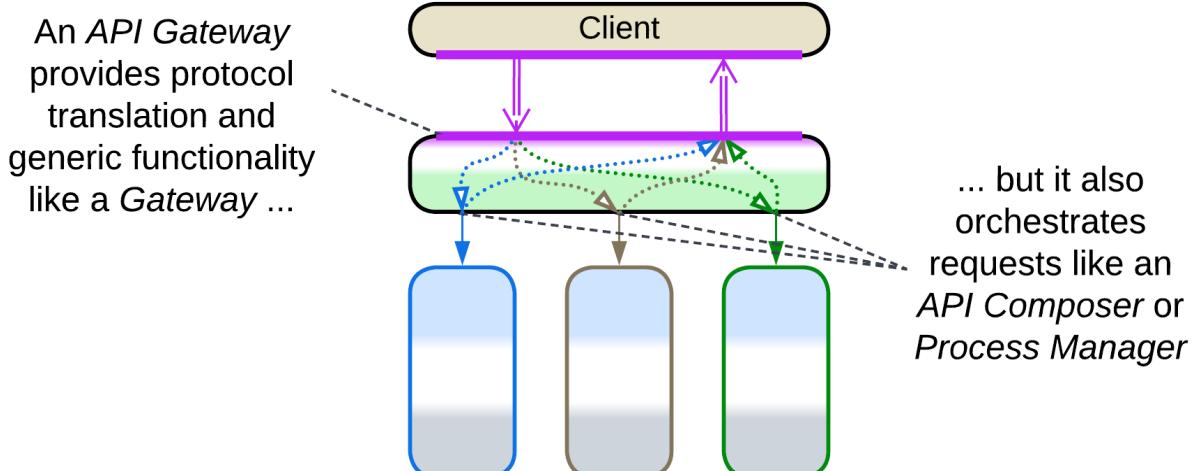
An [Integration Service](#) is a full-scale service (often with a dedicated database) that runs high-level scenarios while delegating the bulk of the work to several other services (remarkably, delegating to a single component forms [Layers](#)). Though an *Integration Service* usually features both functions of *Orchestrator*, in a [control system](#) its *Mediator* role is more prominent while in [processing software](#) it is going to behave more like the *Facade*. A system with an *Integration Service* often resembles a shallow [Top-Down Hierarchy](#).

Example: Order Service in [MP] seems to fit the description.

Variants of composite patterns

Several composite patterns involve an *Orchestrator* and are dominated by its behavior:

API Gateway



An *API Gateway* [MP] is a component that processes client requests (and encapsulates an implementation of a client protocol(s)) as a [Gateway](#) (a kind of [Proxy](#)) but also splits every client request into multiple requests to internal services as an *API Composer* or *Process Manager* (which are *Orchestrators*). It is a common pattern for backend solutions as it provides all the means to isolate the stable core of the system's implementation from its

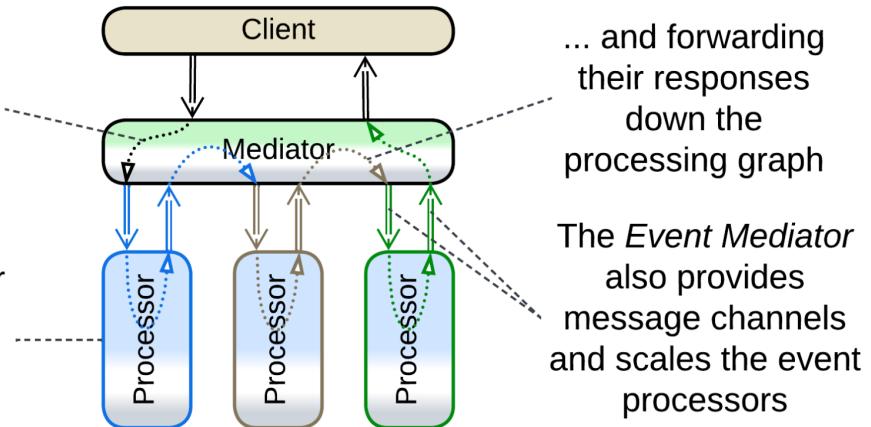
fickle clients. Usually a third-party framework implements and collocates both its aspects, namely *Proxy* and *Orchestrator*, thus simplifying deployment and improving latency.

Example: a thorough article from [Microsoft](#).

Event Mediator

The *Event Mediator* builds a *Pipeline* by transferring client requests to event processors ...

Each event processor contains business logic but does not know the event flow



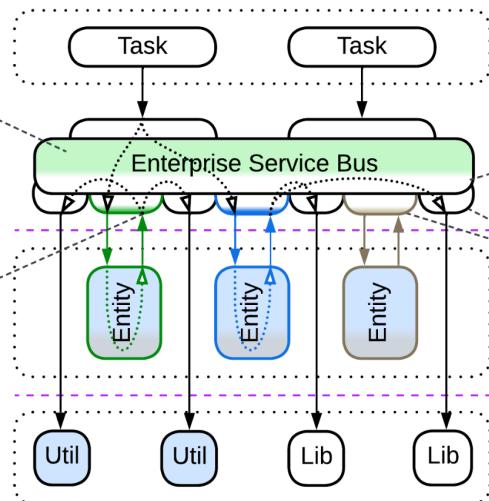
Event Mediator [[FSA](#)] is an *orchestrating Middleware*. It not only receives requests from clients and turns each request into a multistep use case (as a *Process Manager*) but it also manages the deployed instances of services and acts as a medium which transports requests to the services and receives confirmations from them. Moreover, unlike an ordinary *Middleware*, it seems to be aware of all of the kinds of messages in the system and which service each message must be forwarded to, resulting in overwhelming complexity concentrated in a single component which does not even follow the principle of separation of concerns. [[FSA](#)] recommends building a hierarchy of *Event Mediators* from several vendors, further complicating the architecture.

Example: Mediator Topology in the chapter of [[FSA](#)] on Event-Driven Architecture.

Enterprise Service Bus (ESB)

An *Enterprise Service Bus (ESB)* mediates all calls and messages between services

A service posts its request to the *ESB* which knows how to handle it and which services are involved



... and forwarding their responses down the processing graph

The *Event Mediator* also provides message channels and scales the event processors

The *ESB* also implements the transport and cross-cutting concerns like logging

It employs an *Adapter* per service so that the services may vary in their protocols

All that makes the *ESB* a complex component which may become a development bottleneck

Enterprise Service Bus (ESB) [[FSA](#)] is an overgrown *Event Mediator* that incorporates lots of *cross-cutting concerns*, including protocol translation for which it utilizes at least one *Adapter* per service. The combination of a central role in organizations and its complexity was among the main reasons for the demise of [Enterprise Service-Oriented Architecture](#).

Example: Orchestration-Driven Service-Oriented Architecture in [[FSA](#)].

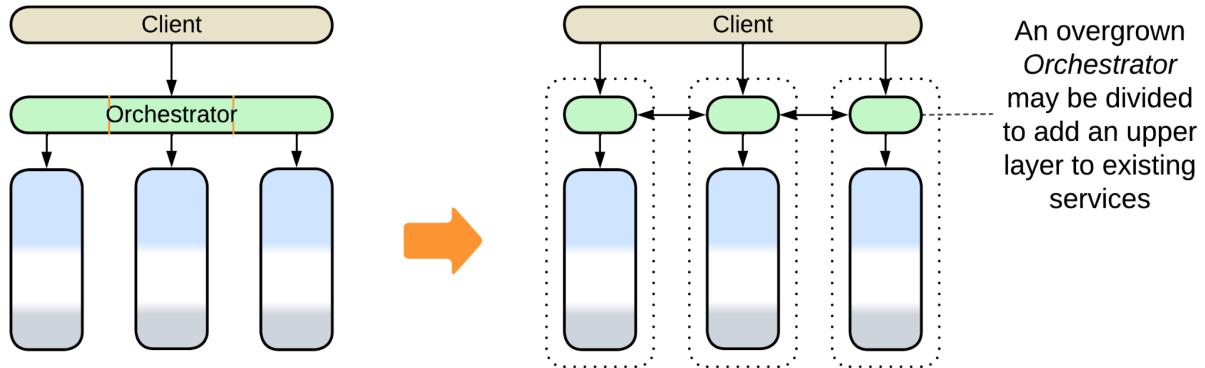
Evolutions

Employing an *Orchestrator* has two pitfalls:

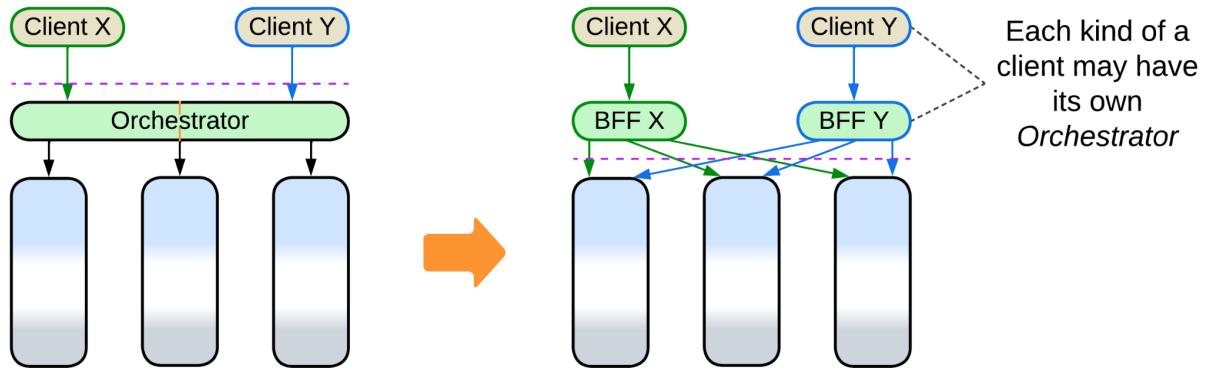
- The system becomes slower because too much communication is involved.
- A single *Orchestrator* may grow too large and rigid.

There is one way to counter the first point and more than one to solve the second:

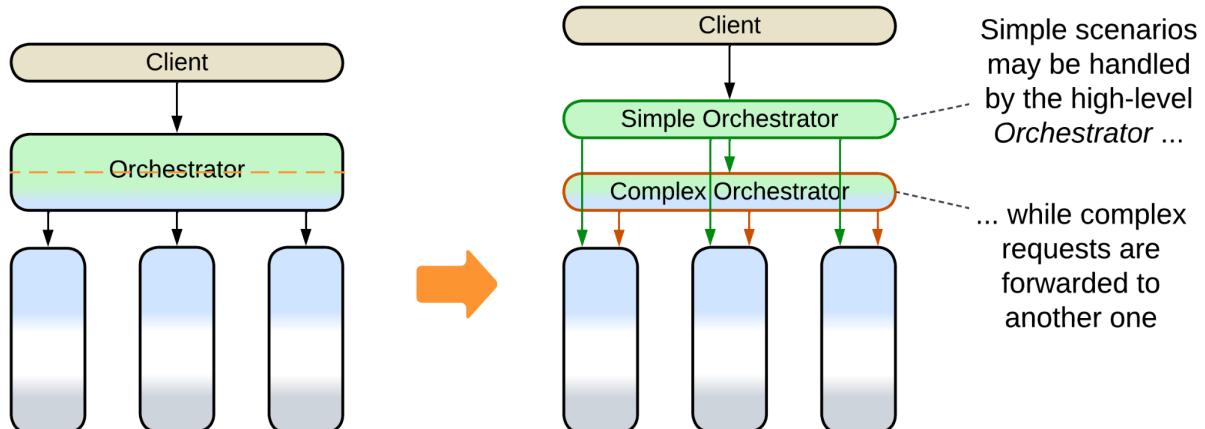
- Subdivide the *Orchestrator* by the system's subdomains, forming [Layered Services](#) and minimizing network communication.



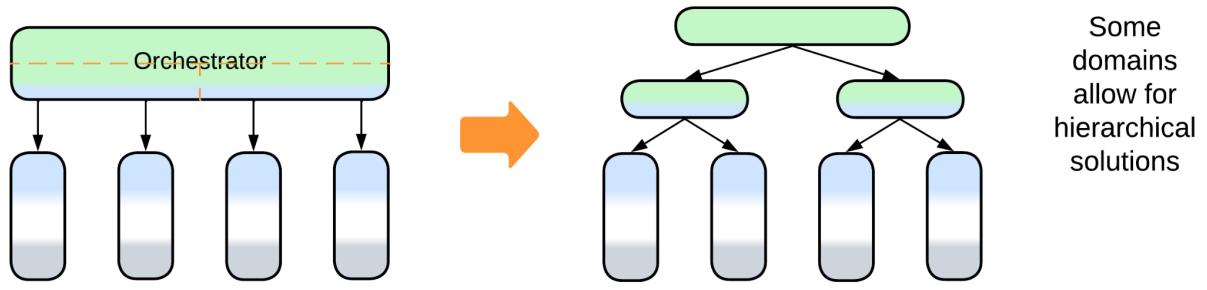
- Subdivide the *Orchestrator* by the type of client, forming [Backends for Frontends](#).



- Add another [layer](#) of orchestration.



- Build a [Top-Down Hierarchy](#).



Summary

An *Orchestrator* distills the high-level logic of your system and keeps it together in a layer which integrates other components. However, the separation of business logic into two layers results in much communication which impairs performance.

Combined Component

Jack of all trades. Use third-party software which covers multiple concerns.

Aspects: those of the individual components it combines.

Variants:

- Message Bus [[EIP](#)],
- API Gateway [[MP](#)],
- Event Mediator [[FSA](#)],
- Persistent Event Log / Shared Event Store,
- Front Controller [[SAHP](#) but not PEAA],
- [Enterprise Service Bus \[FSA\]](#),
- Service Mesh [[FSA](#)],
- Middleware of Space-Based Architecture [[SAP](#), [FSA](#)].

Structure: Two or more (usually) extension patterns combined into a single component.

Type: Extension.

Benefits	Drawbacks
Works off the shelf	Is yet another technology to learn
Improved latency	May not be flexible enough for your needs
Less DevOps effort	May become overcomplicated

References: Mostly [[FSA](#)], [Microsoft](#) on API Gateway, [[DEDS](#)] on Shared Event Store.

Two or three metapatterns may be blended together into a *Combined Component* which is usually a ready-to-use framework which tries to cover (and subtly create) as many project needs as possible to make sure it will never be dropped from the project. On one hand, such a framework may provide a significant boost to the speed of development. On the other – it is going to force you into its own area of applicability and keep you bound within it.

Performance

A *Combined Component* tends to improve performance as it removes the network hop(s) and data serialization between the components it replaces. It is also likely to be highly optimized. However, that matters if you really need all the functionality you are provided with, otherwise you may end up running a piece of software which is too complex and slow for the tasks at hand.

Dependencies

A *Combined Component* has all the dependencies of its constituent patterns.

Applicability

Combined patterns work well for:

- *Series of similar projects.* If your team is experienced with the technology and knows its pitfalls, it will be used efficiently and safely.
- *Small- to medium-sized domains.* An off-the-shelf framework relieves you of infrastructure concerns. No thinking, no decisions, no time wasted.

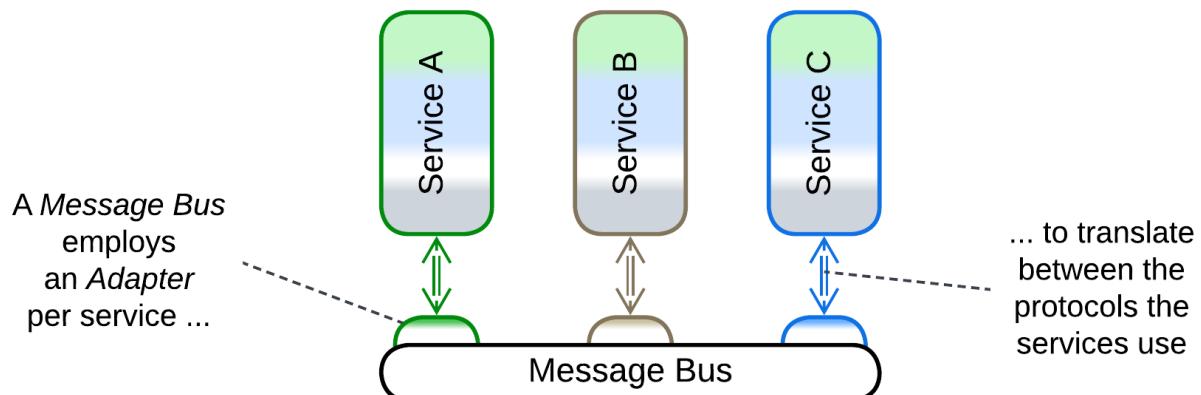
Combined patterns are better avoided in:

- *Research and development*. You may find that the technology chosen is too limiting or a wrong fit for your needs after it has already been deeply integrated into your code and infrastructure.
- *Large projects*. Most of the combined patterns include an *Orchestrator* which tends to grow unmanageably large (requiring [some kind of division](#)) as the project advances.
- *Long-running projects*. You are very likely to step right into *vendor lock-in*. The industry will evolve, leaving you with the obsolete technology you have chosen and integrated.

Variants

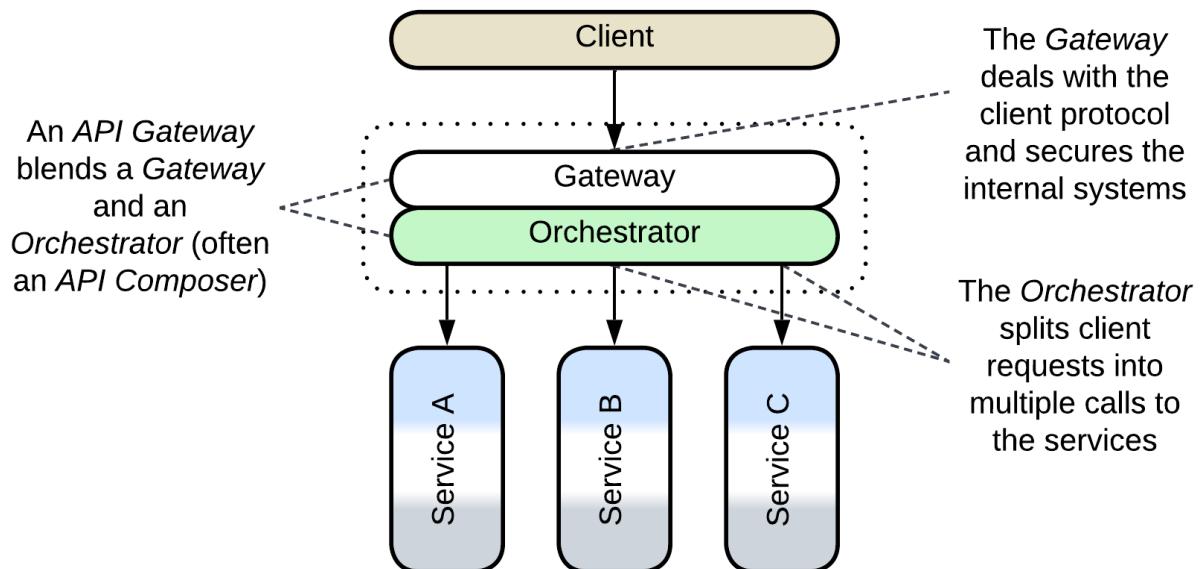
Combined components vary in their structure and properties:

Message Bus



A *Message Bus* [EIP] is a [Middleware](#) which employs an *Adapter* per *service* allowing services that differ in protocols to intercommunicate.

API Gateway



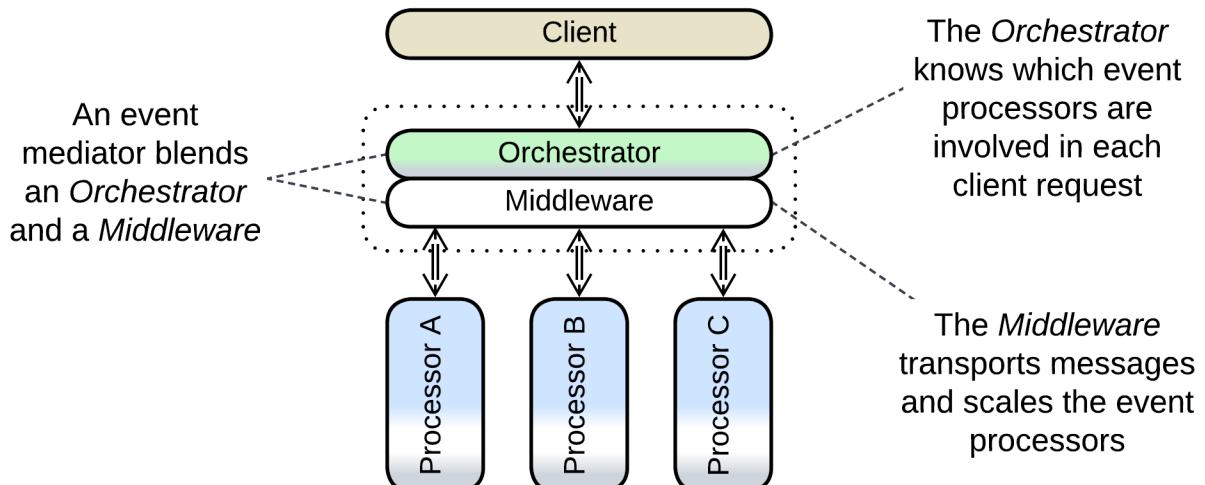
API Gateway [MP] is a component which processes client requests (and encapsulates the client protocol(s)) much like a [Gateway](#) (a kind of [Proxy](#)) but it also splits each client

request into multiple requests to the underlying services like an [API Composer](#) or a [Process Manager \(Orchestrators\)](#).

If the orchestration logic of an *API Gateway* needs to become more complex, it makes sense to split it into a separate *Gateway* and *Orchestrator*, rewriting the latter as a custom [Application Service](#). When there are multiple types of clients that strongly differ in workflows and protocols, one *API Gateway* per client type is used, resulting in [Backends for Frontends](#). A [Cell-Based Architecture](#) relies on *API Gateways* for isolation of its [Cells](#).

Example: a thorough article from [Microsoft](#).

Event Mediator



An *Event Mediator* [FSA] is an *orchestrating Middleware*. It not only receives requests from clients and turns each request into a multi-step use case (as does [Orchestrator](#)) but it also manages instances of the services and acts as the medium which calls them. Moreover, it seems to be aware of all the kinds of messages in the system and which service each message must be forwarded to, resulting in an overwhelming complexity concentrated in a single component which does not even follow the separation of concerns principle. [FSA] proposes countering that by using multiple *Event Mediators* [split over the following dimensions](#):

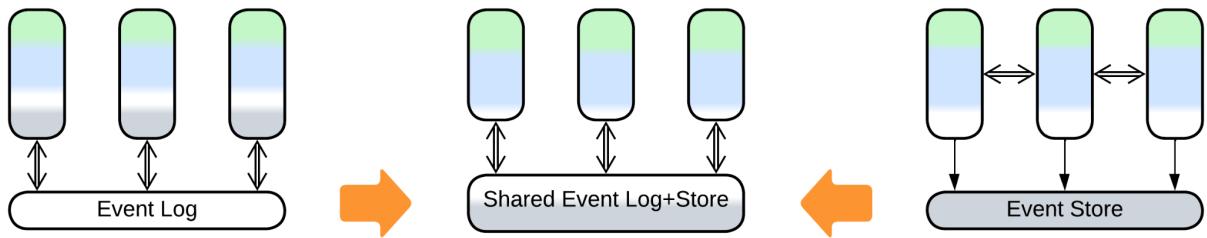
- Client applications or bounded contexts, dividing the *Event Mediator's* responsibility by subdomain.
- Complexity of a use case, with simple scenarios handled by a simple first-line *Event Mediator* and the more complicated scenarios being forwarded to second- and third-line *Event Mediators* which employ advanced [orchestration engines](#).

Either case, strangely, results in multiple *Middlewares* connected to the same set of [services](#).

The *Event Mediator* pattern seems to be well established but, obviously, it may become quite messy for larger projects with nontrivial interactions. Such cases may also be solved by separating the *Middleware* from the *Orchestrator* and [dividing the latter](#) into [Backends for Frontends](#) or [Top-Down Hierarchy](#).

Example: Mediator Topology in the [FSA] chapter on Event-Driven Architecture.

Persistent Event Log, Shared Event Store



When a [Middleware](#) grants long-term persistence, it becomes a [Shared Repository](#). This may happen to an *Event Log* which implements interservice communication.

On the other hand, a shared *Event Store*, which persists changes of internal state of services and replaces the services' databases, may be extended to store interservice events, taking the role of a [Middleware](#).

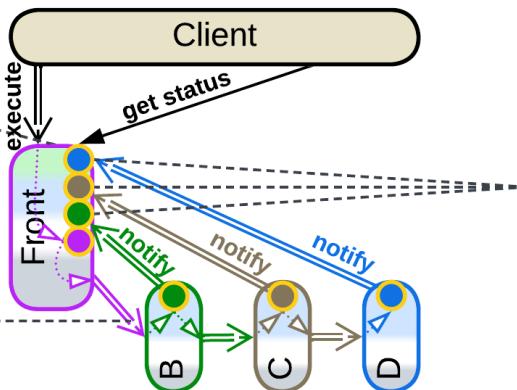
Either case makes changing the event schema troublesome because either all the services that read the event will need to support multiple (historical) schemas or you'll have to overwrite the entire event history, translating the old schema into a new one.

Example: [\[DEDS\]](#) shows this implemented with Apache Kafka.

Front Controller

A *Front Controller* faces client's commands and queries

After doing its part it passes a command down the pipeline

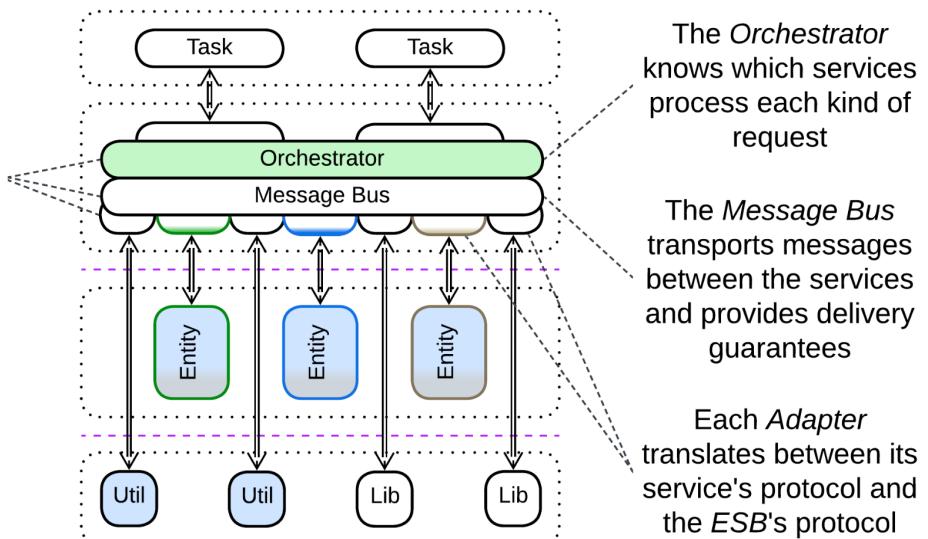


It collects status reports from other services to answer the client's queries

Front Controller [[SAHP](#) but [not PEAA](#)] is the name for the first (client-facing) service of a *pipeline* in [Choreographed Event-Driven Architecture](#) when that service collects information about the status of each request it has processed and forwarded down the *pipeline*. The status is received by listening for notifications from the downstream services and is readily available for the *Front Controller*'s clients, resembling the function of [Query Service \(Polyglot Persistence\)](#) and [Application Service \(Orchestrator\)](#).

Enterprise Service Bus (ESB)

An Enterprise Service Bus contains an *Orchestrator*, a *Middleware*, and also an *Adapter* per service



The *Orchestrator* knows which services process each kind of request

The *Message Bus* transports messages between the services and provides delivery guarantees

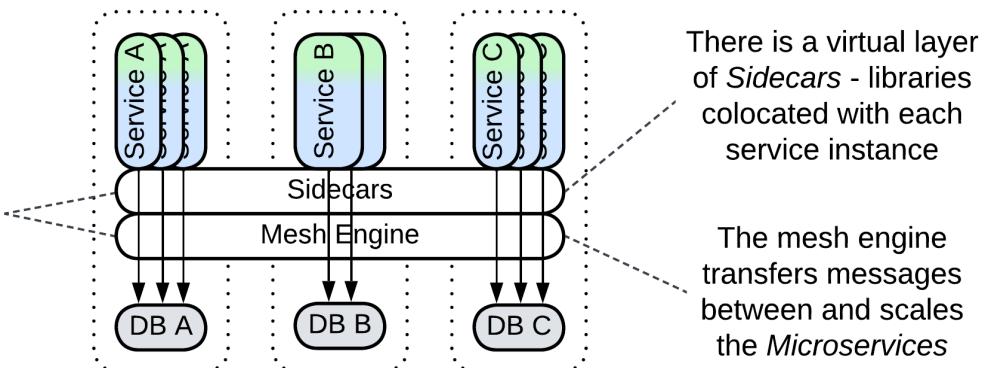
Each *Adapter* translates between its service's protocol and the *ESB's* protocol

An [Enterprise Service Bus](#) (ESB) [FSA] is a mixture of *Message Bus* (with a stack of *Adapters* per service) and *Event Mediator* (built-in *Orchestrator*) which turns this kind of *Middleware* into the core of the system. The combination of a central role in organizations and its complexity was among the main reasons for the demise of [Enterprise Service-Oriented Architecture](#).

Example: Orchestration-Driven Service-Oriented Architecture in [FSA], [how it is born](#) and [how it dies](#) by Neal Ford.

Service Mesh

Service Mesh is a distributed *Middleware* for *Microservices*



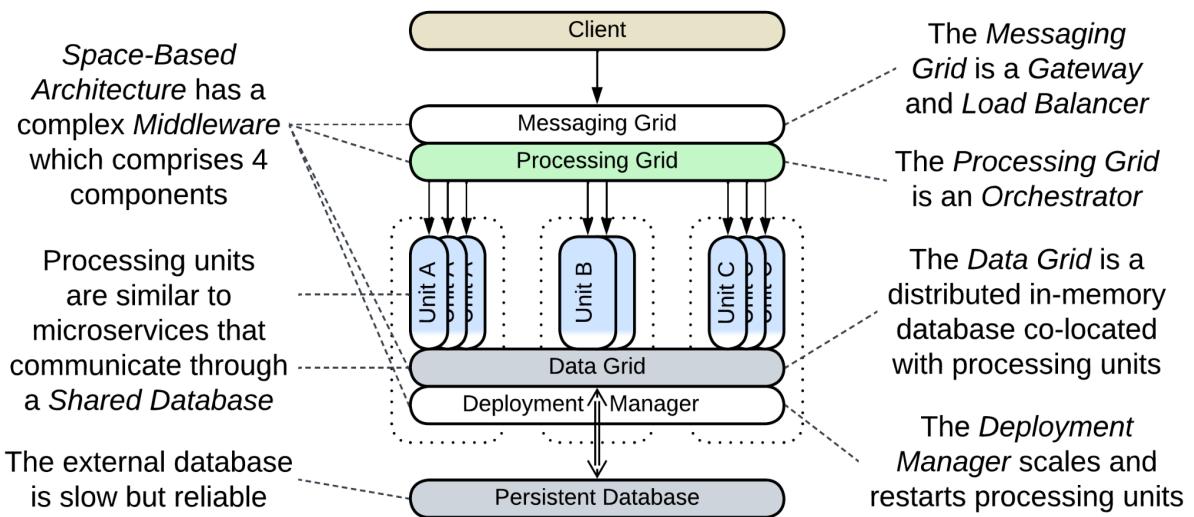
There is a virtual layer of *Sidecars* - libraries colocated with each service instance

The mesh engine transfers messages between and scales the *Microservices*

A Service Mesh [FSA] is a *Middleware* that employs one or more *Sidecars* [DDS] per service as a place for *cross-cutting concerns* (logging, observability, encryption, protocol translation). A service accesses its *Sidecar* without performance and stability penalties as they are running on the same machine. The totality of deployed *Sidecars* makes a system-wide virtual layer of shared libraries: though the *Sidecars* are physically separate, they are often identical and stateless, so that a service that accesses one *Sidecar* may be thought of as accessing all of them.

A Service *Mesh* is also responsible for dynamic scaling (it creates new instances if the load increases and destroys them if they become idle) and failure recovery of the services. Last but not least, it provides a messaging infrastructure for the [Microservices](#) to intercommunicate.

Middleware of Space-Based Architecture



[Space-Based Architecture \[SAP, FSA\]](#) relies on the most complex *Middleware* known – it incorporates all four of the *extension metapatterns*:

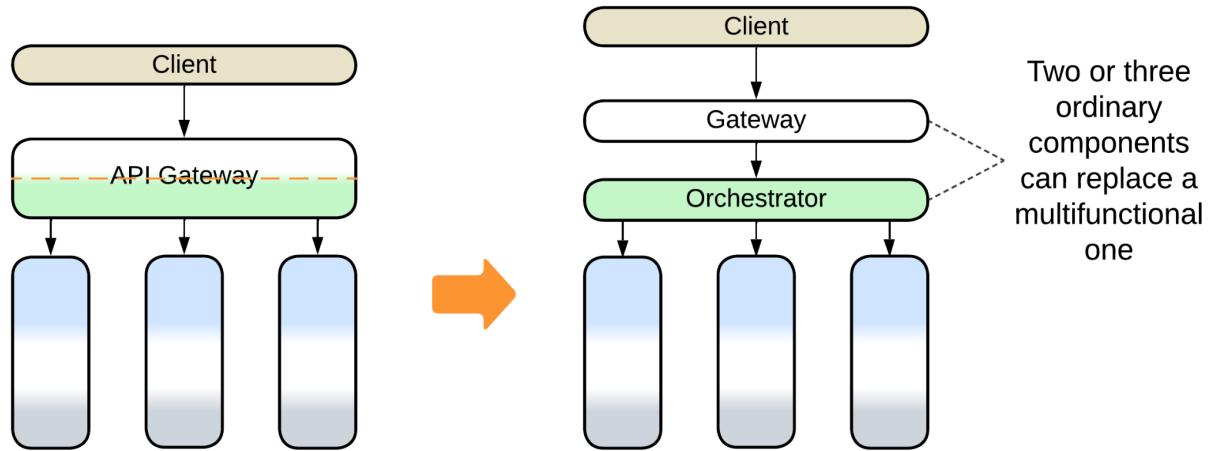
- The *Messaging Grid* is a [Proxy](#) (combination of [Gateway](#), [Dispatcher](#), and [Load Balancer](#)) that receives, preprocesses, and persists client requests. Simple requests are forwarded to a less loaded *Processing Unit* (service with business logic) while complex ones go to the *Processing Grid*.
- A *Processing Grid* is an [Orchestrator](#) which manages multi-step workflows for complex requests.
- A [Data Grid](#) is a [distributed](#) in-memory [database](#). It is built of caching nodes which are co-located with instances of *Processing Units*, making the database access extremely fast. However, the speed and scalability is paid for with stability – any data in RAM is prone to disappearing. Therefore the *Data Grid* backs up all the changes to a slower *Persistent Database*.
- A *Deployment Manager* is a [Middleware](#) that creates and destroys instances of *Processing Units* (which are paired to the nodes of the *Data Grid*), just like [Service Mesh](#) does for [Microservices](#) (which are paired to [Sidecars \[DDS\]](#)). However, in contrast to *Service Mesh*, it does not provide a messaging infrastructure because *Processing Units* communicate by [sharing data](#) via the *Data Grid*, not by sending messages.

The four layers of the *Space-Based Architecture's Middleware* are reasonably independent. Together they make a system that is both more scalable and more complex than *Microservices*.

Evolutions

The patterns that involve [orchestration](#) (*API Gateway*, *Event Mediator* and *Enterprise Service Bus*) may allow for most of the evolutions of [Orchestrator](#) by deploying multiple instances of the *Combined Component* which differ in orchestration logic. There is also one unique evolution:

- Replace the *Combined Component* with several specialized ones



Summary

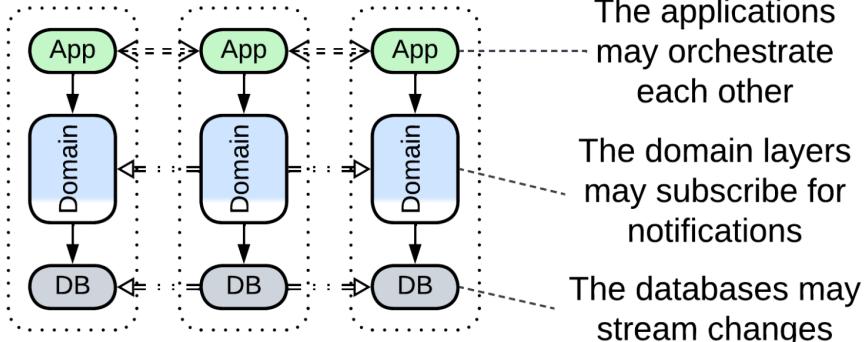
A *Combined Component* is a ready-to-use framework that may speed up development but will likely constrain your project to follow its guidelines with no regard to your real needs.

Part 4. Fragmented Metapatterns

There are several patterns with no system-wide layers. Some of them incorporate two or three orthogonal domains which vary in abstractness to the extent that a service (limited to a subdomain) of one domain acts as a layer for another domain.

Layered Services

Layering separates the interacting parts from the independent parts of the services

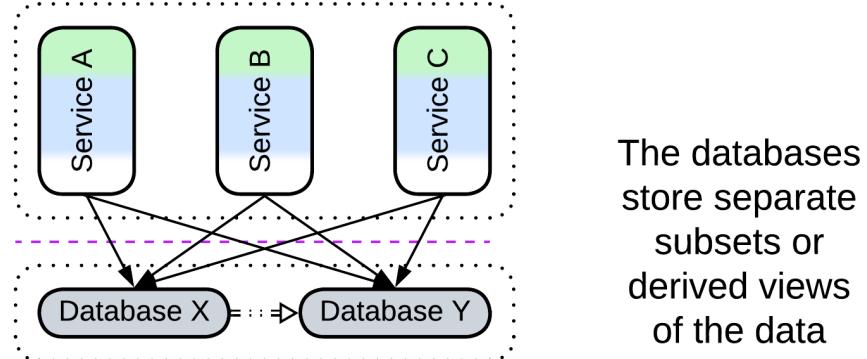


Layered Services is an umbrella metapattern that highlights implementation details of *Services*, *Pipeline*, or *Monolith*.

Includes: Orchestrated Three-Layered Services, Choreographed Two-Layered Services, Command Query Responsibility Segregation (CQRS).

Polyglot Persistence

The system gains benefits from multiple database technologies

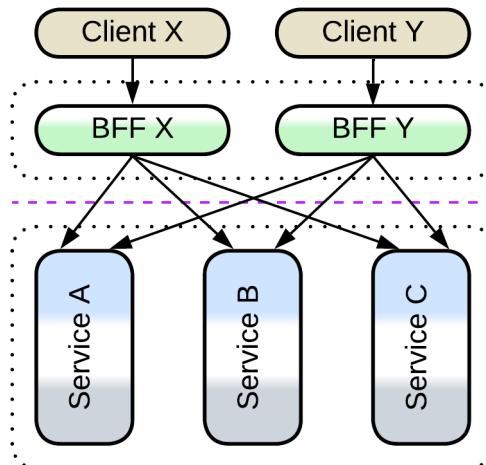


Polyglot Persistence is about using multiple data stores which differ in roles or technologies. Each of the upper-level components may have access to any data store. Each data store is a *Shared Repository*.

Includes: specialized databases, private and shared databases, data file, Content Delivery Network (CDN); read-only replica, Reporting Database, CQRS View Database, Memory Image, Query Service, search index, historical data, Cache-Aside.

[Backends for Frontends](#)

Each BFF is dedicated to its kind of client



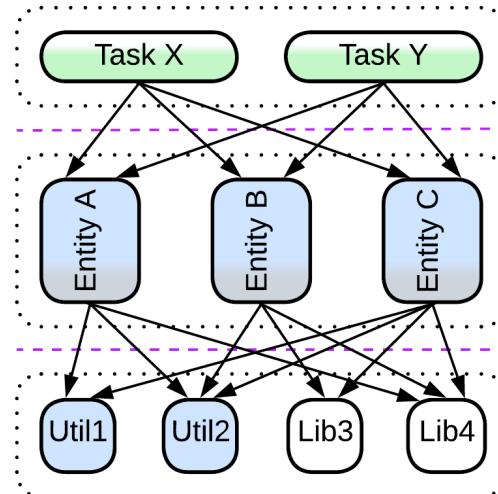
Each BFF orchestrates all the services

Backends for Frontends feature a service (*BFF*) for each kind of the system's client. A *BFF* may be a *Proxy*, *Orchestrator* or both. Each *BFF* communicates with all the components below it. The pattern looks like multiple *Proxies* or *Orchestrators* deployed in parallel.

Includes: Layered Microservice Architecture.

[Service-Oriented Architecture](#)

SOA is 3 or 4 layers of services



Each task orchestrates entities

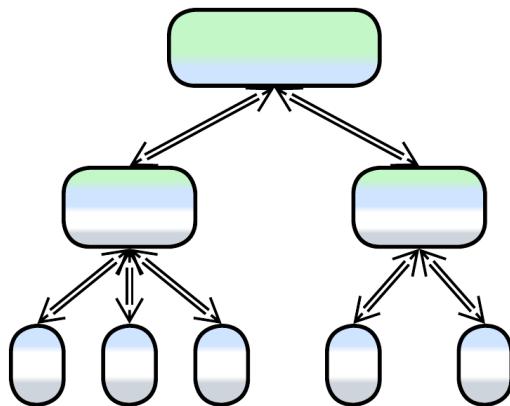
Entities use libraries and utilities

SOA comprises three or four layers of services, with each layer making a domain. The upper layer contains *Orchestrators* which are often client-specific, just like *BFFs*. The second layer incorporates business rules and is divided into business subdomains. The lower layer(s) are libraries and utilities, grouped by functionality and technologies. Any component may use (orchestrate) anything below it.

Includes: Segmented Architecture; distributed monolith, enterprise SOA.

Hierarchy

Hierarchy is a tree of components



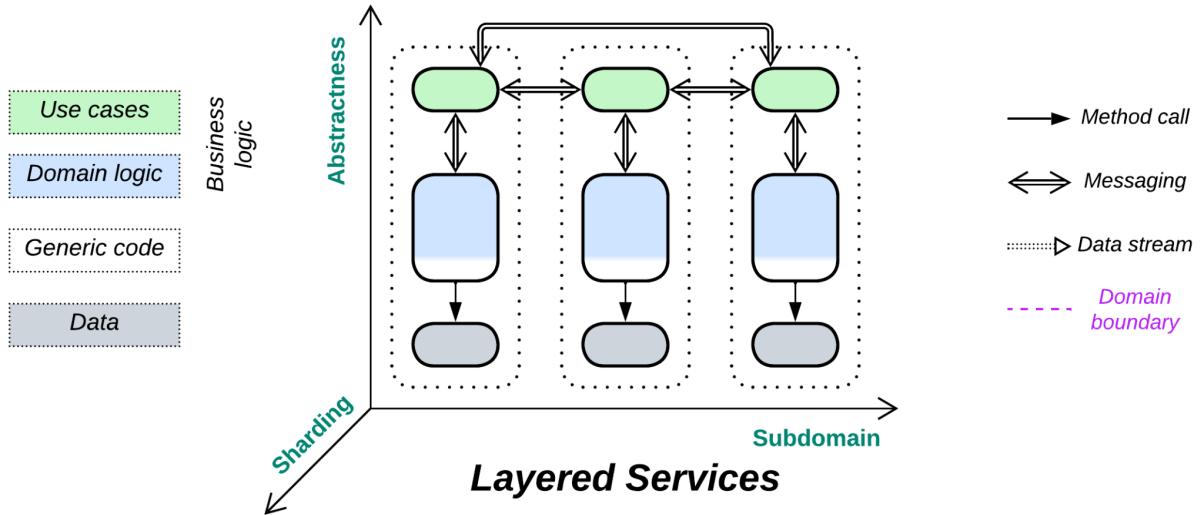
Each root orchestrates its children

Child nodes may be specialized or polymorphic

Some domains allow for hierarchical composition where the functionality is spread throughout a tree of components.

Includes: Orchestrator of Orchestrators, Presentation-Abstraction-Control (PAC) and Hierarchical Model-View-Controller (HMVC), Bus of Buses, and Cell-Based (Microservice) Architecture (WSO2 version) (Services of Services).

Layered Services



Cut the cake. Divide each service into layers.

Variants:

- [Orchestrated](#) Three-Layered Services,
- [\(Pipelined\) Choreographed](#) Two-Layered Services,
- [\(Pipelined\)](#) Command Query Responsibility Segregation (CQRS) [[MP](#), [LDDD](#)].

Structure: Subdomain services divided into layers.

Type: Implementation of [Services](#), [Pipeline](#) or [Monolith](#), correspondingly.

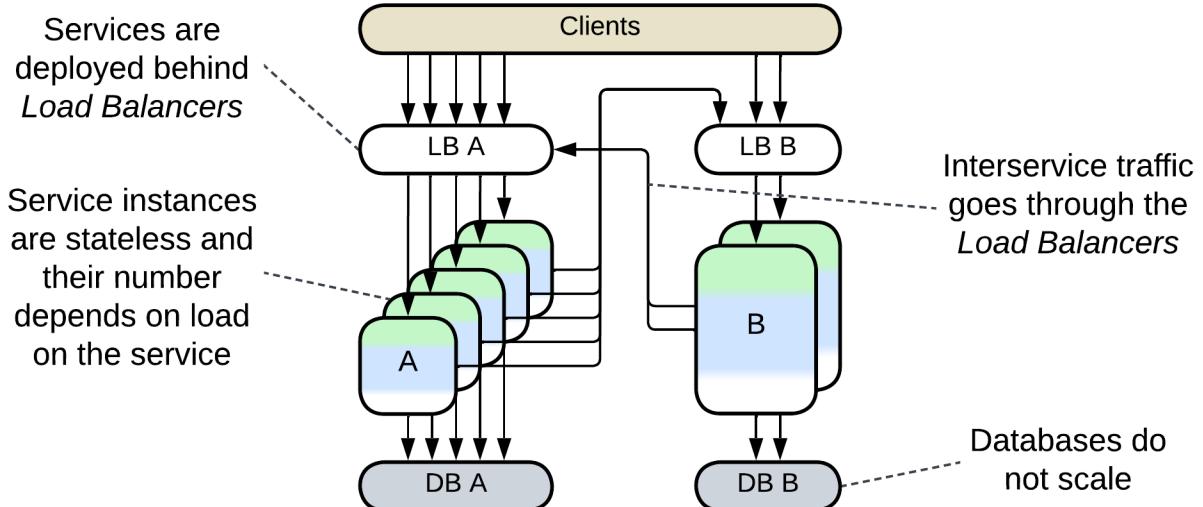
Layered Services is an umbrella architecture for common implementations of systems of [Services](#). It does not introduce any special features as layers are completely encapsulated by the service which they belong to. Still, as the services may communicate at different layers, there are a couple of things to learn by exploring the subject matter.

Performance

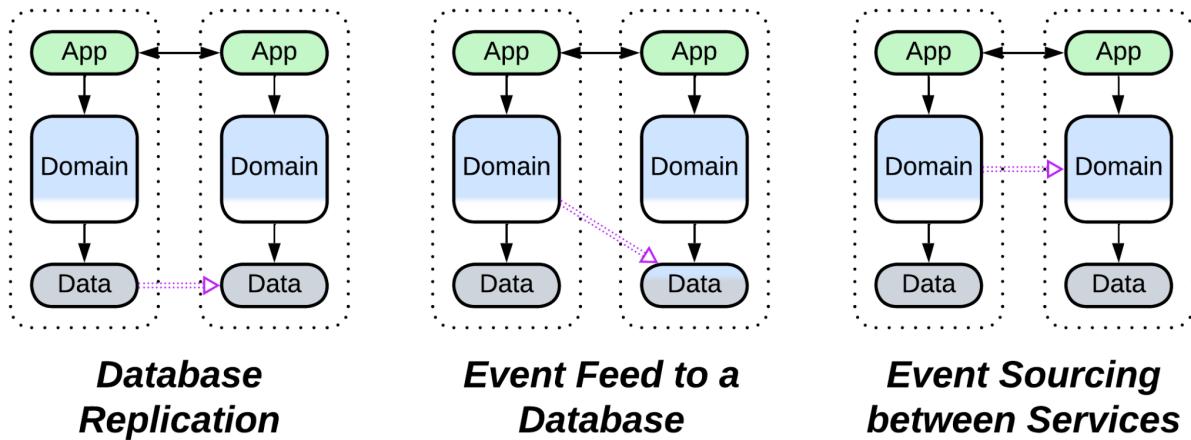
Layered Services are similar to [Services](#) performance-wise: use cases that involve a single service are the fastest, those that need to synchronize states of multiple services are the slowest.

Remarkable features of *Layered Services* include:

- Independent scaling of layers of the services. It is common to have multiple [instances](#) (with the number varying from service to service and changing dynamically under load) of the layers that contain business logic while the corresponding data layers (databases) are limited to a single instance.



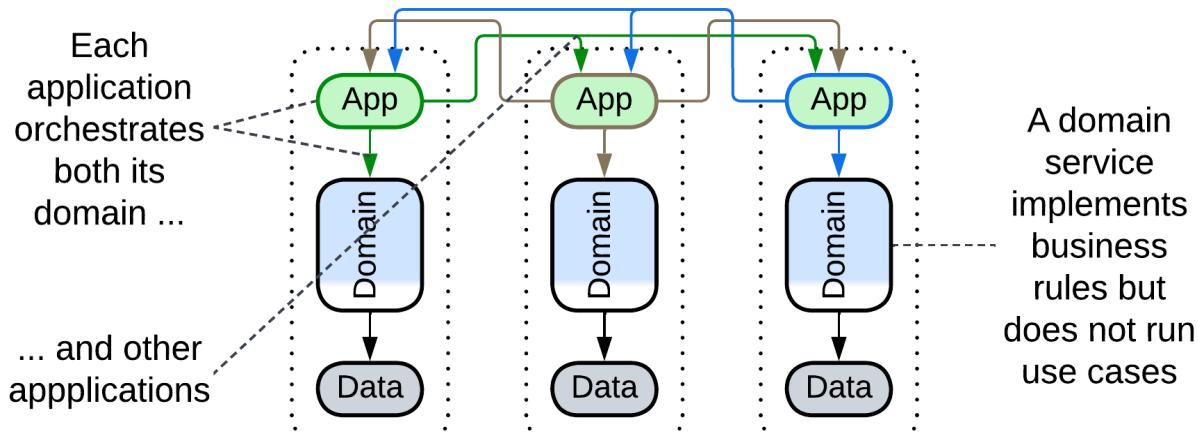
- The option to establish additional communication channels between lower layers in order to drive [CQRS](#) databases ([read/write replicas](#) of the same database) or [CQRS Views](#) (cached subsets of data from other services) [[MP](#)].



Variants

Layered Services vary in the number of [layers](#) and in the layer through which the [services](#) communicate:

Orchestrated Three-Layered Services

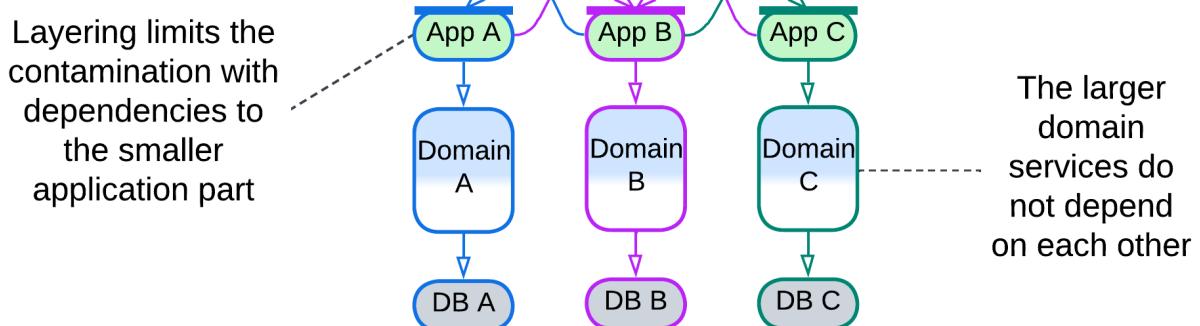


Probably the most common backend architecture has three layers: *application*, *domain*, and *infrastructure* [DDD]. The application layer orchestrates the domain layer.

If such an architecture is divided into services, each of them receives a part of every layer, including application, which means that now there are as many *Orchestrators* as services. Each *Orchestrator* implements the API of its service by integrating (calling or messaging into) the domain layer of its service and APIs of other services, which makes all the *Orchestrators* interdependent:

Dependencies

The upper (application) layer of each service orchestrates both its middle (domain) layer and the upper layers of other services, resulting in mutual orchestration and interdependencies.



The good thing is that the majority of the code belongs to the domain layer which depends only on its databases. The bad thing is that changes in the application of one service may affect the application layers of all of the other services.

Relations

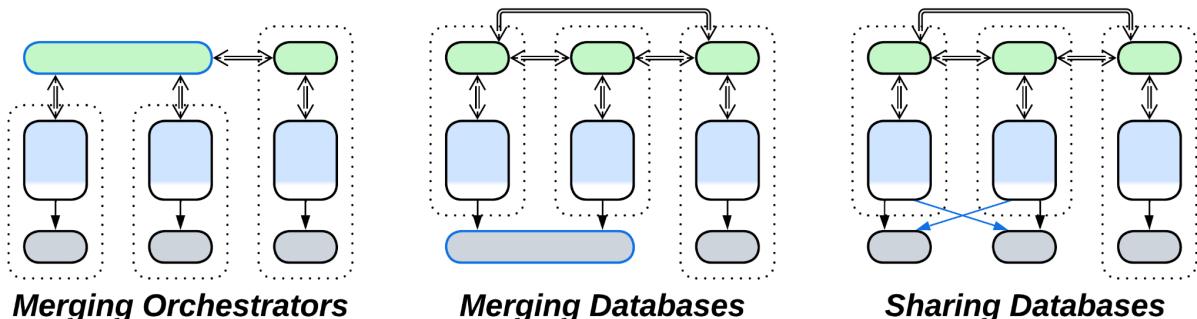
Three-layered services:

- Implement Services.
- Are derived from Layers and Services.
- Have multiple Integration (sub)Services (Orchestrators).

Evolutions

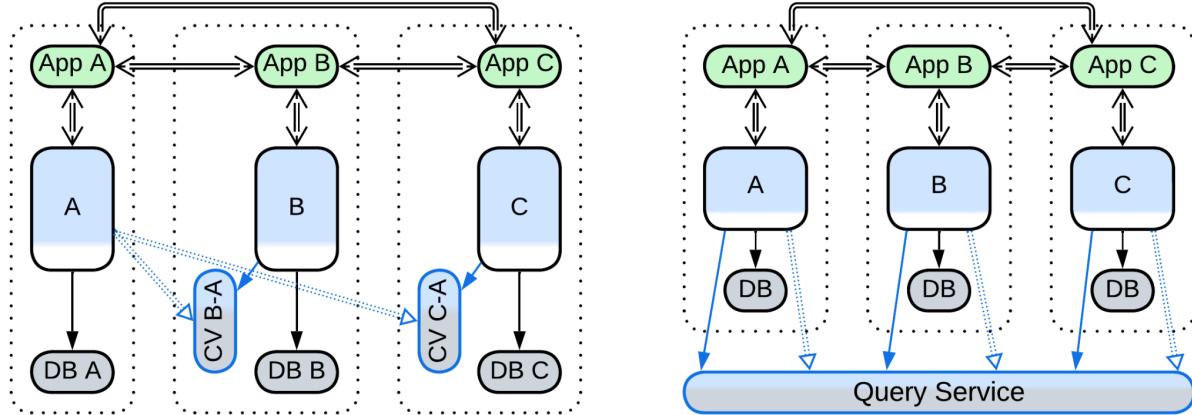
Orchestrated Layered Services may become coupled, which is resolved either by merging their layers:

- A part of or the whole application layer can be merged into a shared Orchestrator.
- Some or all the *databases* can be united into a Shared Database or shared as Polyglot Persistence.



or by building derived datasets:

- A [CQRS View \[MP\]](#) inside a service aggregates any events from other services which its owner is interested in.
- A dedicated [Query Service \[MP\]](#) captures the whole system's state by subscribing to events from all the services.

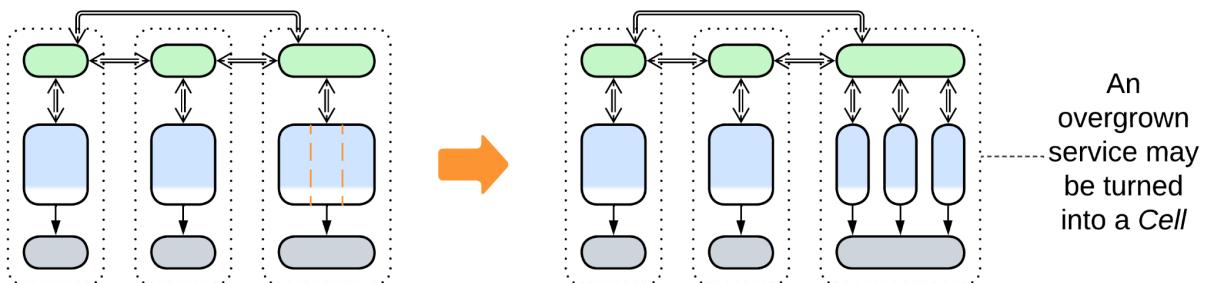


CQRS Views

Query Service

If the services become too large:

- The middle layer can be split into [Cells](#).

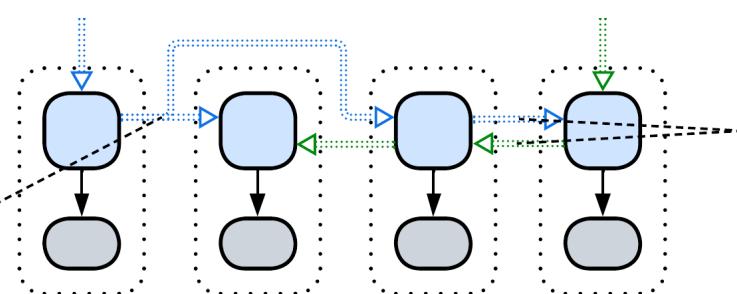


Choreographed Two-Layered Services

Two-layered services make a system of pipelines

The high-level logic resides in the topology of event channels

Use cases may run in different directions



If there is no [orchestration](#), there is no role for the [application](#) layer. [Choreographed](#) systems are made up of services that implement individual steps of request processing. The sequence of actions (*integration logic*) which three-layered systems put in the [Orchestrators](#) now moves to the graph of *event channels* between the services. This means that with choreography the high-level part of the business logic (use cases) exists outside of the code of the constituent services.

Dependencies

Dependencies are identical to those of a [Pipeline](#) or [choreographed Services](#) except that each service also depends on its database.

Relations

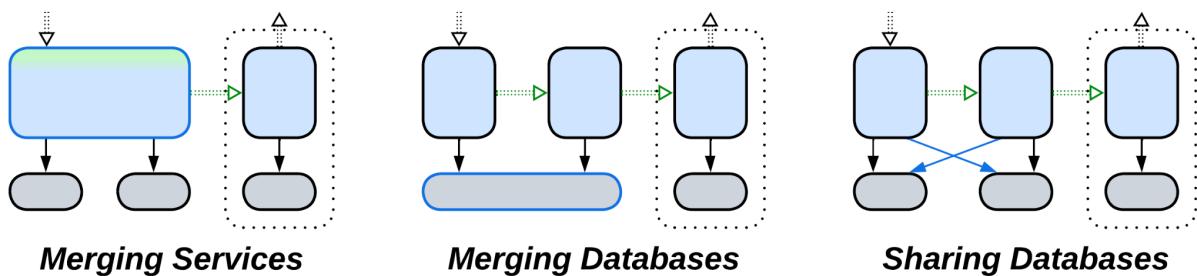
Two-layered services:

- Implement [Pipeline](#).
- Are derived from [Layers](#) and [Pipeline](#).

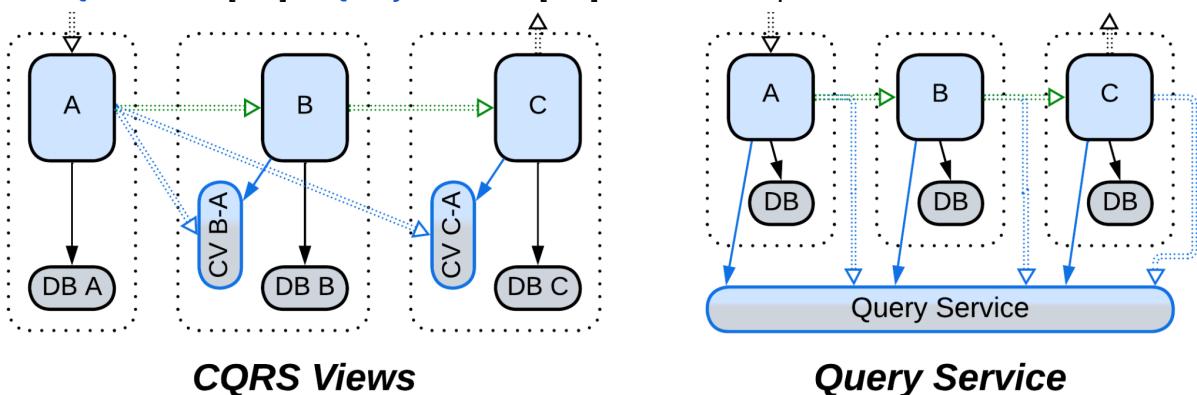
Evolutions

If *Choreographed Layered Services* become coupled:

- The *business logic* of two or more services can be merged together, resulting in [Polyglot Persistence](#).
- Some databases can be united into a [Shared Database](#) or shared as [Polyglot Persistence](#).

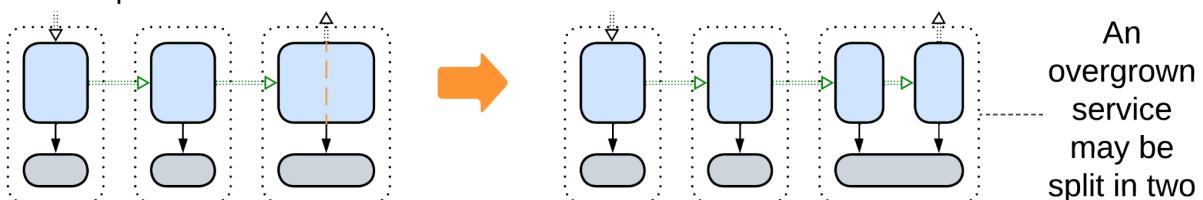


[CQRS Views](#) [MP] or [Query Services](#) [MP] are also an option:

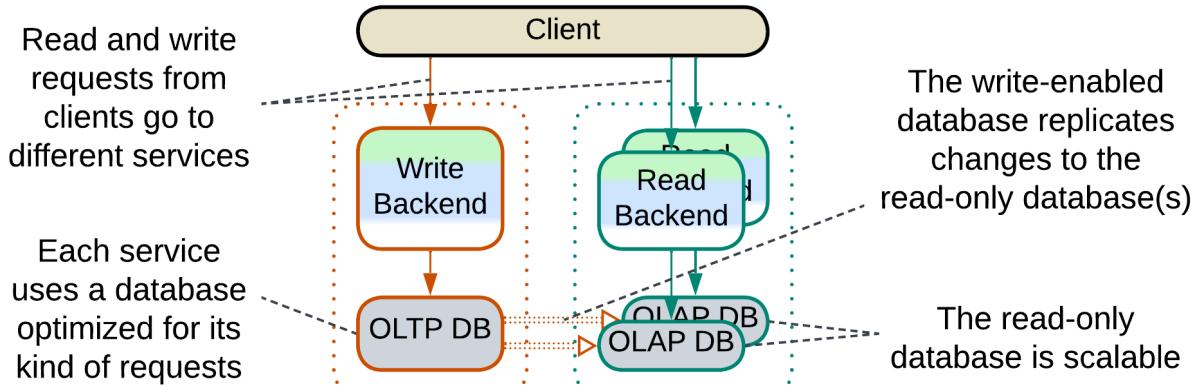


An overgrown service can be:

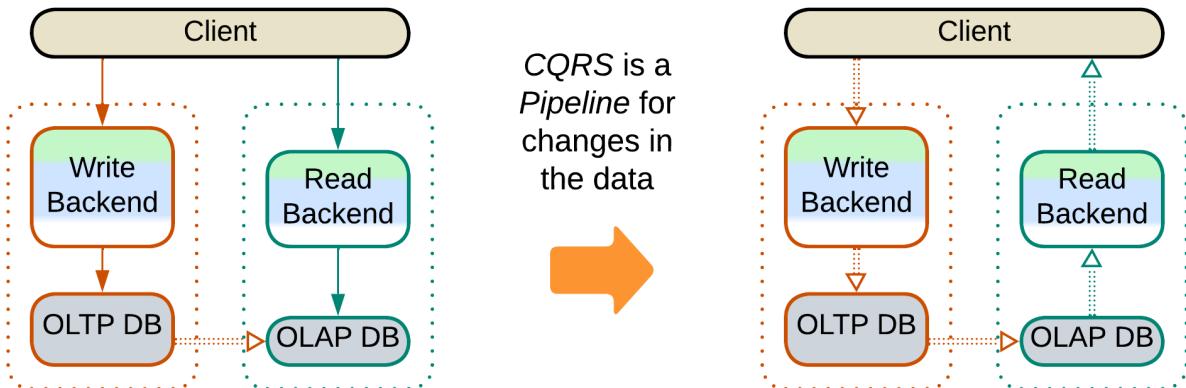
- Split in two



Command Query Responsibility Segregation (CQRS)



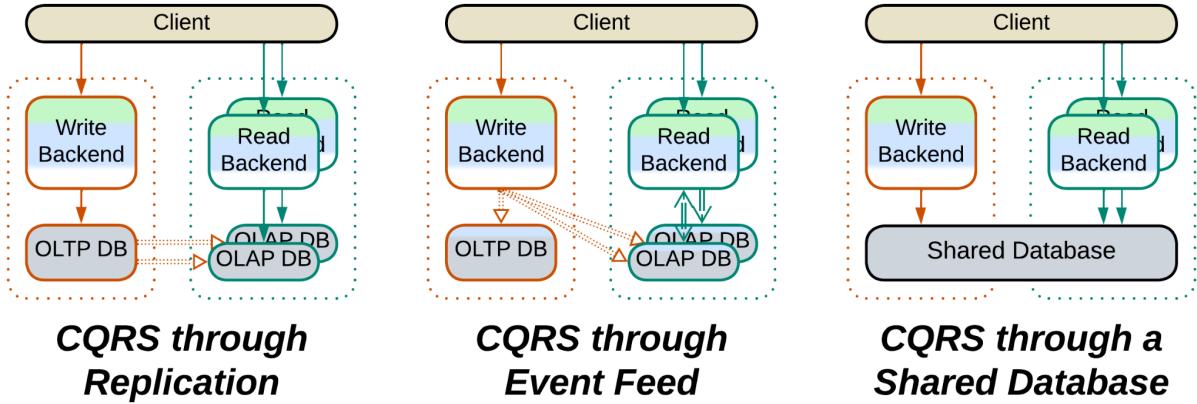
Command Query Responsibility Segregation (CQRS) [[MP](#), [LDDD](#)] is, essentially, the division of a [layered](#) application or a service into two (rarely more) [services](#), one of which is responsible for write access (handling *commands*) to the domain data while the other(s) deal with read access (*queries*), thus [creating](#) a data [pipeline](#) (see the diagram below). Such an architecture makes sense when the write and read operations don't rely on a common vision (*model*) of the domain, for example, writes are individual changes ([OLTP](#)) that require cross-checks and validation of input while reads show aggregated data ([OLAP](#)) and may take long time to complete (meaning that [forces](#) for the read and write paths differ). If there is nothing to share in the code, why not separate the implementations?



This separation brings in the pros and cons of [Services](#): commands and queries may differ in technologies (including database schemas or even types), forces, and teams at the expense of [consistency](#) (database replication delay) and increasing the system's complexity. In addition, for read-heavy applications the read database(s) is easy to scale.

CQRS has several variations:

- The database may be shared, commands and queries may use dedicated databases, or the read service may maintain a [Memory Image / Materialized View](#) [[DDIA](#)] fed by events from the write service (as in other kinds of *Layered Services*).
- Data [replication](#) may be implemented as a [pipeline](#) between the databases (based on nightly snapshots or [log-based replication](#)) or a [direct event feed](#) from the OLTP code to the OLAP database.

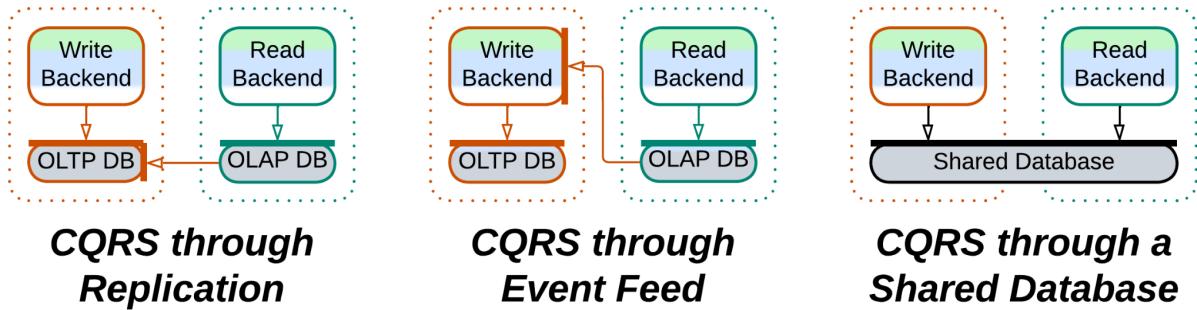


It is noteworthy that while ordinary *Layered Services* usually communicate through their upper-level components that drive use cases, a CQRS system is held together by spreading data changes through its lowest layer.

Examples: Martin Fowler has a [short article](#) and Microsoft a [longer one](#).

Dependencies

Each backend depends on its database (its technology and schema). The OLTP to OLAP data replication requires an additional dependency that corresponds to the way the replication is implemented:



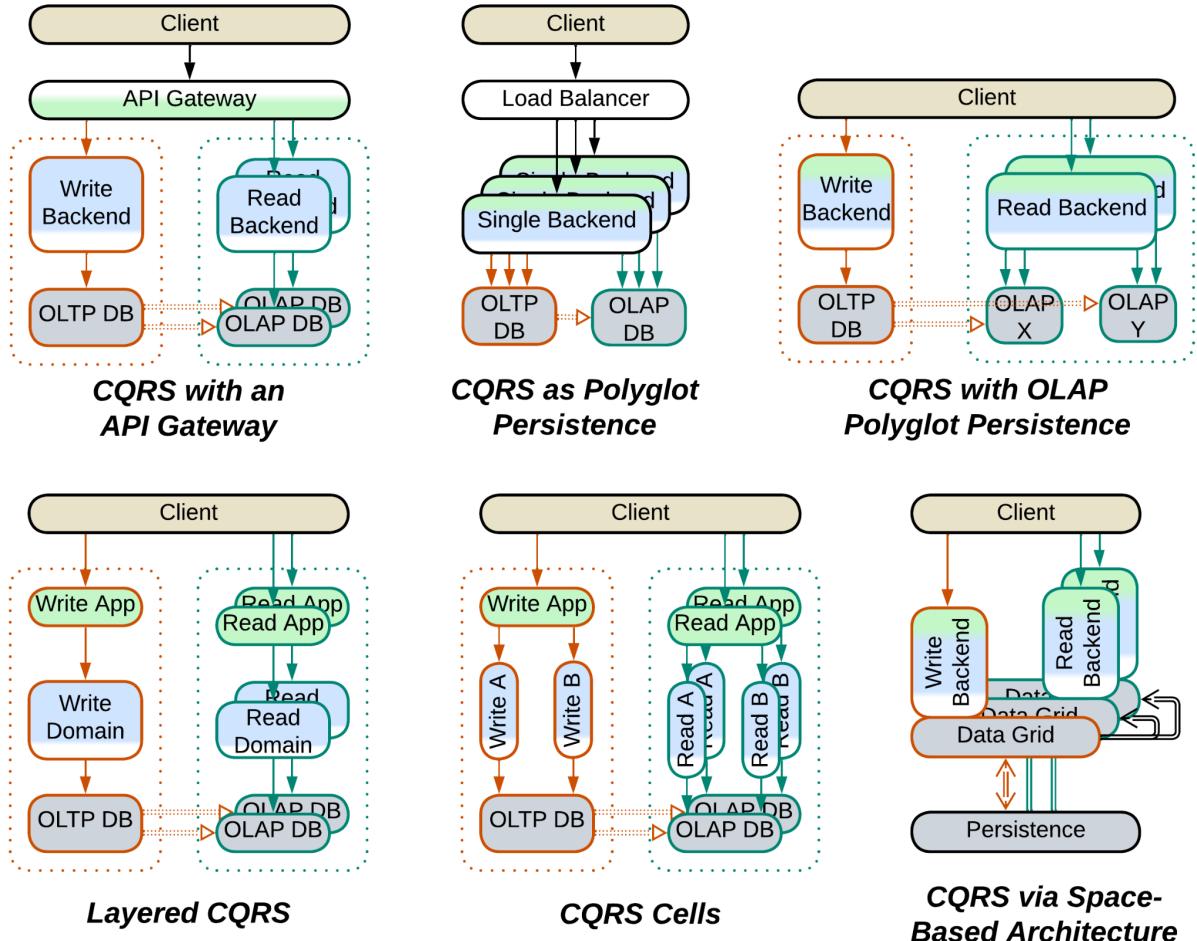
Relations

CQRS:

- Implements [Monolith](#) (a whole system or a single [service](#)).
- Is derived from [Layers](#) and [Pipeline](#).
- Is a development of [Polyglot Persistence](#).

Evolutions

- You will usually need a [Reverse Proxy](#) or an [API Gateway](#) to segregate commands from queries.
- If the commands and queries become intermixed, the business logic can be merged together but the databases are left separate, resulting in [Polyglot Persistence](#).
- Both read and write backends can be split into [Layers](#) or [Services](#) (yielding [Cells](#)).
- Applying [Space-Based Architecture](#) may further improve performance.
- Multiple schemas or even kinds of OLAP databases can be used simultaneously ([Polyglot Persistence](#)).

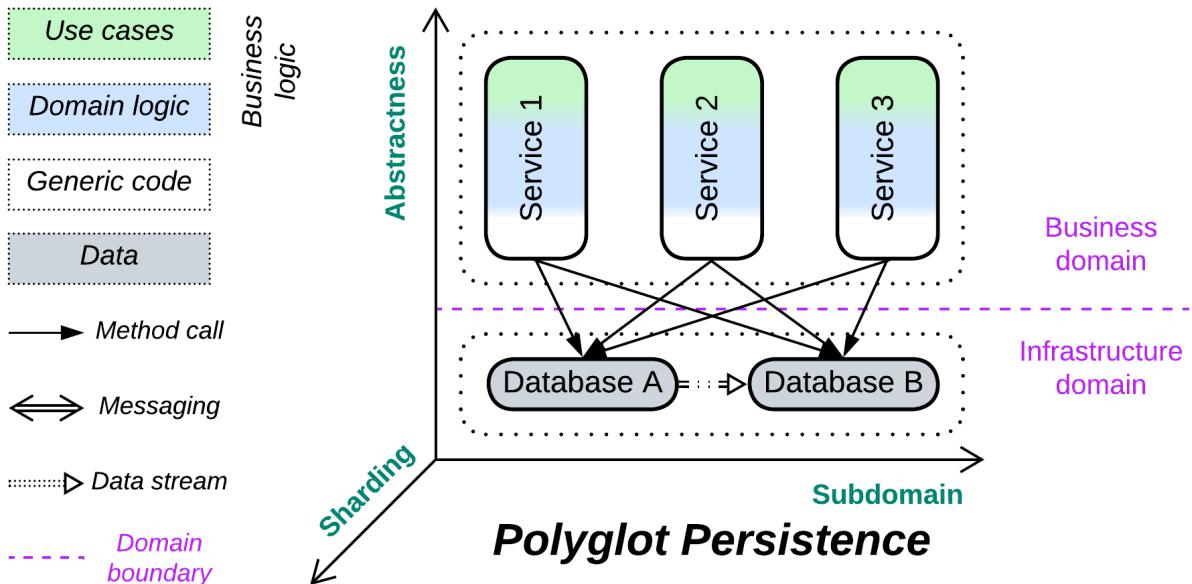


Summary

Layered Services is an umbrella pattern that conjoins:

- *Three-Layered Services* where each service [orchestrates](#) other services.
- *Two-Layered Services* that form a [Pipeline](#).
- CQRS that separates read and write request processing paths.

Polyglot Persistence



Unbind your data. Use multiple specialized databases.

Known as: Polyglot Persistence.

Aspects: those of the [databases](#) involved.

Variants:

Independent storage:

- Specialized Databases,
- Private and Shared Databases,
- Data File / Content Delivery Network (CDN).

Derived storage:

- Read-Only Replica,
- [Reporting Database](#) / CQRS View Database [[MP](#)] / Event-Sourced View [[DEDS](#)] / Source-Aligned (Native) Data Product Quantum (DPQ) of Data Mesh [[SAHP](#)],
- [Memory Image](#) / Materialized View [[DDIA](#)],
- Query Service [[MP](#)] / Front Controller [[SAHP](#)] but not PEAA / Data Warehouse [[SAHP](#)] / Data Lake [[SAHP](#)] / Aggregate Data Product Quantum (DPQ) of Data Mesh [[SAHP](#)],
- External Search Index,
- Historical Data / [Data Archiving](#),
- Database Cache / [Cache-Aside](#).

Structure: A layer of data services used by higher-level components.

Type: Extension, derived from [Shared Repository](#).

Benefits	Drawbacks
Performance is fine-tuned for various data types and use cases	The peculiarities of each database need to be learned
Less load on each database	Much more work for the DevOps team
The databases may satisfy conflicting forces	More points of failure in the system Consistency is hard or slow to achieve

References: The [original](#) and closely related [CQRS](#) articles from Martin Fowler, chapter 7 of [\[MP\]](#), chapter 11 of [\[DDIA\]](#) and much information dispersed all over the Web.

You can choose a dedicated technology for each kind of data or pattern of data access in your system. That improves performance (as each database engine is optimized for a few use cases), distributes load between the databases, and may solve conflicts between forces (like when you need both low latency and large storage). However, you'll likely have to hire several experts to get the best use of and to support the multiple databases. Moreover, having your data spread over multiple databases makes it the application's responsibility to keep the data in sync (by implementing some kind of distributed transactions or making sure that the clients don't get stale data).

Performance

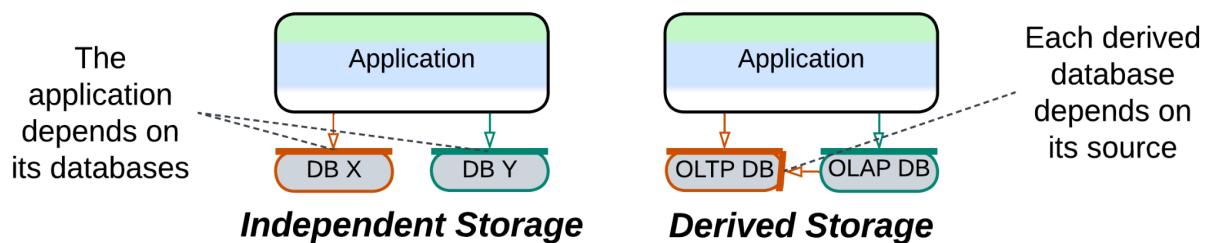
Polyglot Persistence aims at improving performance through the following means:

- Optimize for [specific data use cases](#). It is impossible for a single database to be good at everything.
- Redirect read traffic to [read-only database replicas](#). The write-enabled *leader* database then processes only the write requests.
- [Cache any frequently used data](#) in a fast in-memory database to let the majority of client requests be served without hitting the slower persistent storage.
- Build a [view of the states of other services](#) in the system to avoid querying them.
- Maintain an external [index](#) or [Memory Image](#) for use with tasks that don't need the history of changes.
- [Purge old data](#) to a slower storage.
- Store read-only sequential [data as files](#), often close to the end users who download them.

Read-write separation introduces a [replication lag](#) which is a pain when data consistency is important for the system's clients.

Dependencies

In general, each service depends on all of the databases which it uses. There may also be an additional dependency between the databases if they share a dataset (one or more databases are derived).



Applicability

Polyglot Persistence helps:

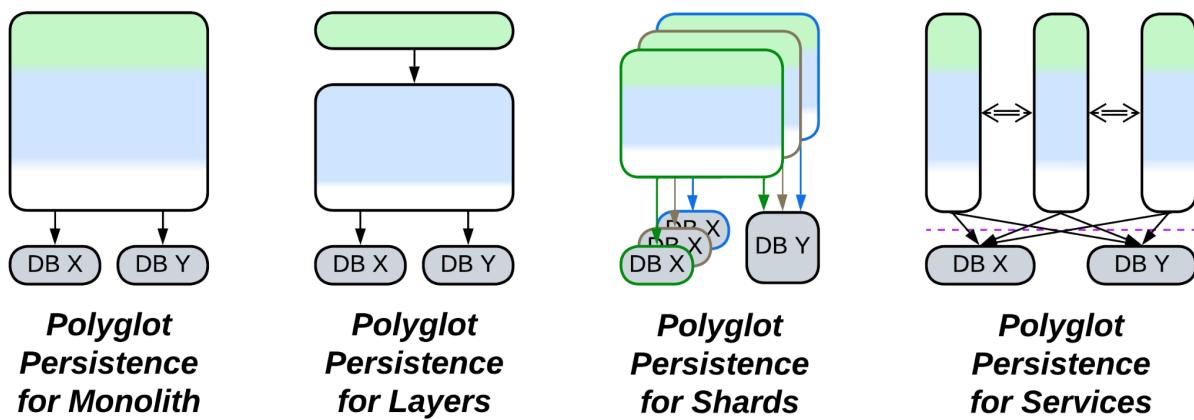
- *High load and low latency projects.* [Specialized Databases](#) shine when given fitting tasks. [Caching](#) and [Read-Only Replicas](#) take the load off the main database. [External Search Indices](#) save the day.

- *Event sourcing.* [Materialized Views](#) maintain the current states of the system's components.
- *Conflicting forces.* An instance of a stateless service inherits many of the qualities of the database which it accesses for any given request it is processing. When there are several databases, the qualities of a service instance may vary from request to request, depending on which database is involved.

Polyglot Persistence may harm:

- *Small projects.* Properly setting up and maintaining multiple databases is not that easy.
- *High availability.* Each database which your system uses will tend to fail in its own crazy way.
- *User experience.* For systems with read-write database separation the replication lag between the databases will make you [choose](#) between reading changes from the *leader* (write database), adding synchronization code to your application to wait for the read database to be updated, and risking returning outdated results to the users.

Relations



Polyglot Persistence:

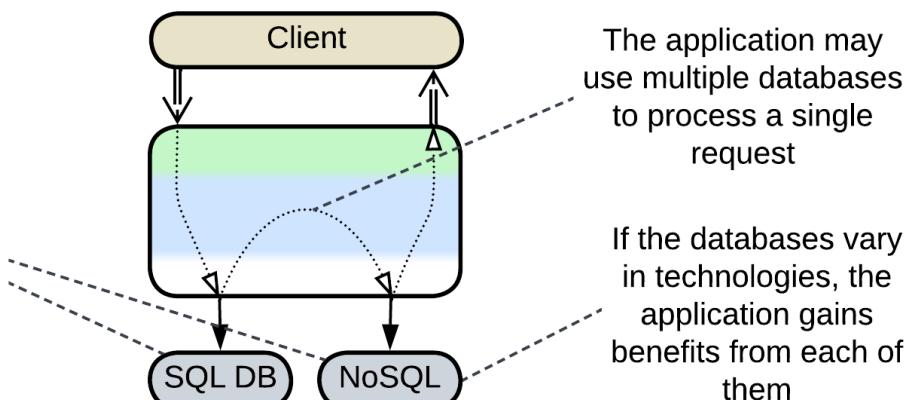
- Extends [Monolith](#), [Shards](#), [Layers](#), or [Services](#).
- Is derived from [Layers](#) (persistence layer) or [Shared Repository](#).
- Variants with derived databases have an aspect of [Pipeline](#) and are closely related to [CQRS](#).

Variants with independent storage

Many cases of *Polyglot Persistence* use multiple datastores just because there is no single technology that matches all the application's needs. The databases used are filled with different subsets of the system's data:

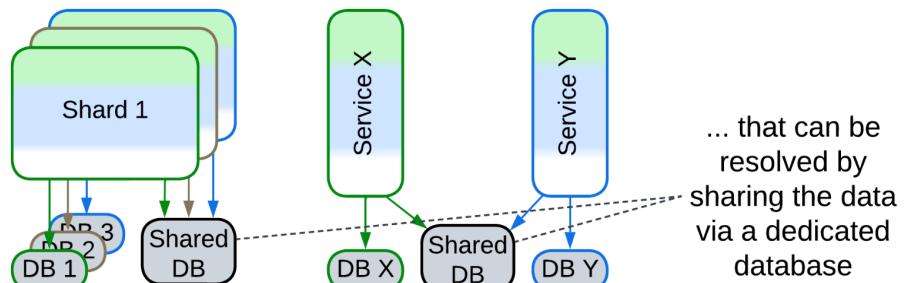
Specialized Databases

Polyglot Persistence is about using several specialized databases



Private and Shared Databases

If shards or services become coupled through a part of the system's data ...

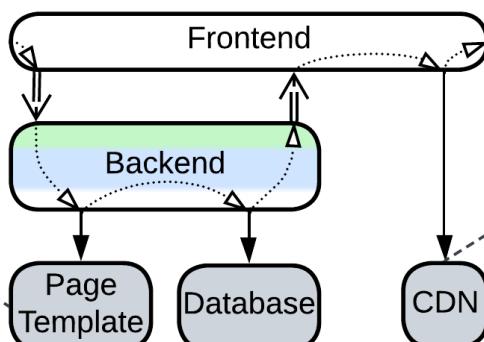


If several services or shards become coupled through a subset of the system's data, that subset can be put into a separate database which is accessible to all the participants. All the other data remains private to the [shards](#) or [services](#).

Data File, Content Delivery Network (CDN)

Web page templates are often stored as ordinary files

Images and videos are also distributed as files via a *Content Delivery Network*



Some data is happy to stay in files. Web frameworks load web page templates from OS files and store images and videos in a *Content Delivery Network (CDN)* which replicates the data all over the world so that each user downloads the content from the nearest server (which is faster and cheaper).

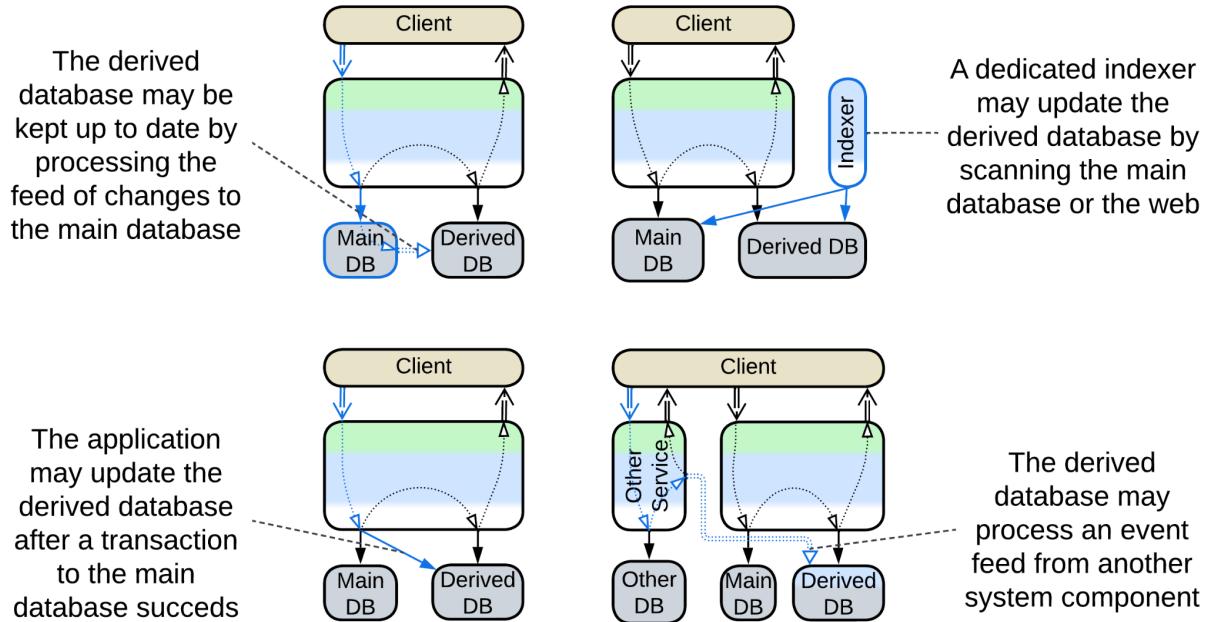
Variants with derived storage

In other cases there is a single writable database (*system of record* [[DDIA](#)]) which is the main source of truth from which the other databases are derived. The primary reason to use

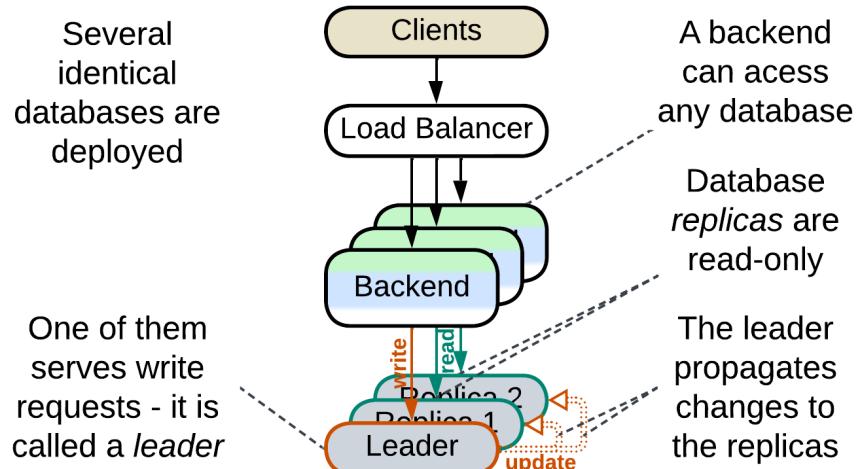
several databases is to [relieve the main database of read requests](#) and maybe support some additional qualities: special kinds of queries, aggregation for [materialized](#) and [CQRS views](#), full text search for [text indices](#), huge dataset size for [historical data](#) or low latency for an [in-memory cache](#).

The updates to the derived databases may come from:

- the main database as [Change Data Capture](#) (CDC) [DDIA] (a log of changes),
- the application after it changes the main database (see caching strategies below),
- another service as [event stream](#) [DDIA, MP],
- a dedicated *indexer* that periodically crawls the main database or web site.



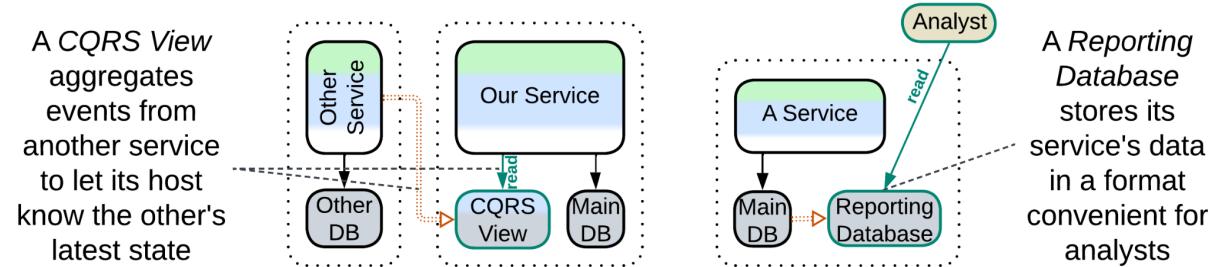
Read-Only Replica



Multiple instances of the database are deployed and one of them is the *leader* [DDIA] instance which processes all writes to the system's data. The changes are then replicated to the other instances (via [Change Data Capture](#) (CDC)) which are used for read requests. Distributing workload over multiple instances increases maximum read throughput which the system is capable of, as the database is usually the system's bottleneck. Having several running [replicas](#) greatly improves reliability and allows for nearly instant recovery of

database failures as any replica may quickly be promoted to the leader role to serve write traffic.

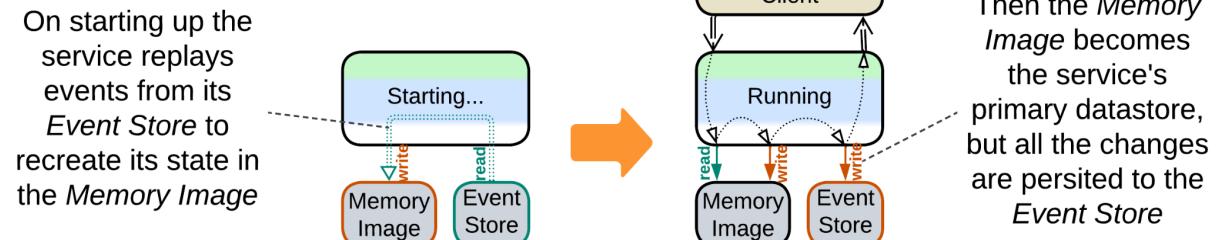
Reporting Database, CQRS View Database, Event-Sourced View, Source-Aligned (Native) Data Product Quantum (DPQ) of Data Mesh



It is common wisdom that a database is good for either *OLTP* (transactions) or *OLAP* (queries). Here we have two databases: one optimized for commands (write traffic protected with transactions) and another one for complex analytical queries. The databases differ at least in schema (*OLAP* schema is optimized for queries) and often vary in type (e.g. SQL vs NoSQL).

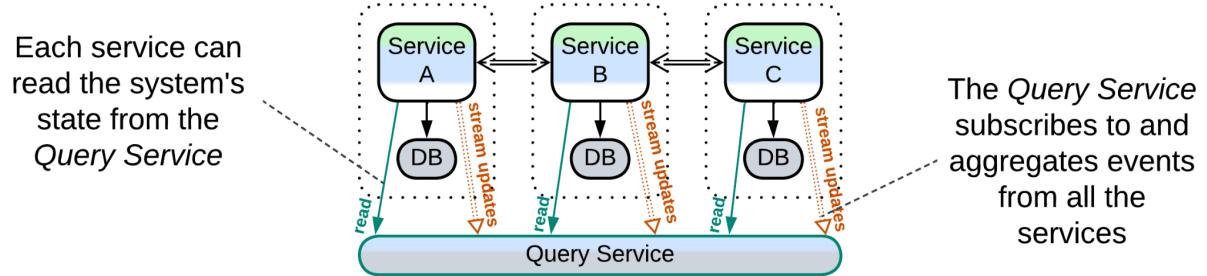
A [Reporting Database](#) (or [Source-Aligned \(Native\) Data Product Quantum](#) of [Data Mesh](#) [[SAHP](#)]) derives its data from a write-enabled database in the same subsystem (service) while a [CQRS View](#) [[MP](#)] or [Event-Sourced View](#) [[DEDS](#)] is fed a stream of events from another service from which it filters the data relevant to its owner. This way a [CQRS View](#) lets its owner service query (its replica of) the data that originally belonged to other services.

Memory Image, Materialized View



Event sourcing (of [Event-Driven Architecture](#) or [Microservices](#)) is all about changes. A service persists only *changes* to its data instead of the *current* data. As a result, the service needs to aggregate its history into a [Memory Image](#) ([Materialized View](#) [[DDIA](#)]) by loading a snapshot and replaying any further events to rebuild its current state (which other architectural styles store in databases) and start operating.

Query Service, Front Controller, Data Warehouse, Data Lake, Aggregate Data Product Quantum (DPQ) of Data Mesh

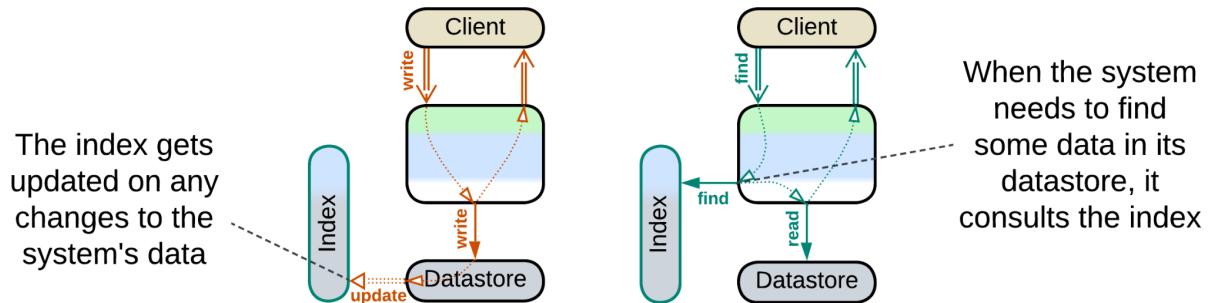


A *Query Service* [MP] (or *Aggregate Data Product Quantum of Data Mesh* [SAHP]) subscribes to events from several full-featured services and aggregates them into its database, making it a *CQRS View* of several services or even the whole system. If any other service or a data analyst needs to process data which belongs to multiple services, it retrieves it from the *Query Service* which has already joined the data streams and represents the join in a convenient way.

A *Front Controller* [SAHP] but *not PEAA*] is a *Query Service* embedded in the first (user-facing) service of a *Pipeline*. It collects status updates from downstream components of the *Pipeline* to track the state of every request being processed by the *Pipeline*.

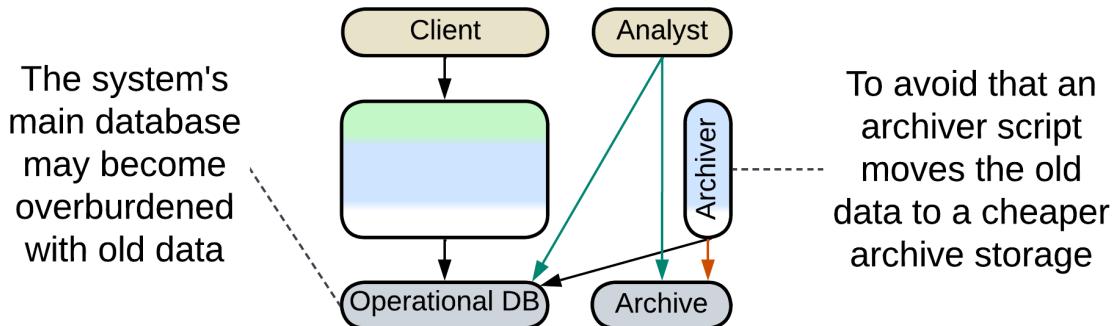
Data Warehouse [SAHP] and *Data Lake* [SAHP] are *analytical* databases that connect directly to and import all the data from the *operational* (main) databases of all the system's services. A *Data Warehouse* translates the imported data into its own unified schema while a *Data Lake* stores the imported data in its original formats.

External Search Index



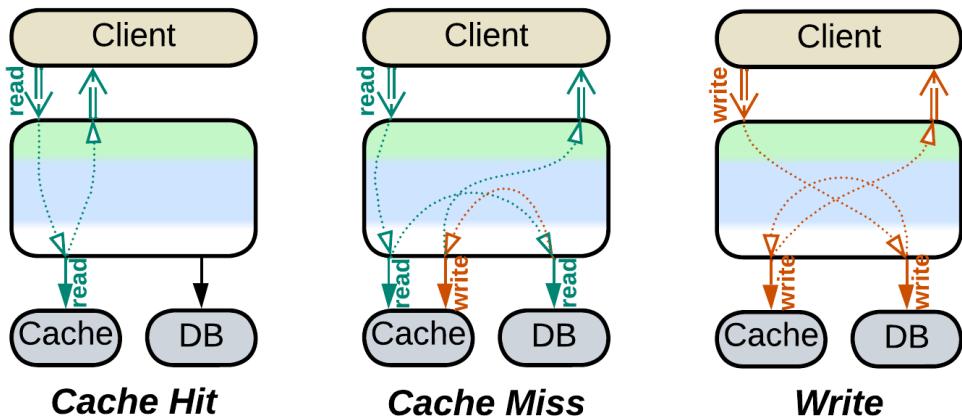
Some domains require a kind of search which is not naturally supported by ordinary database engines. Full text search, especially *NLP*-enabled, is one such case. Geospatial data may be another. If you are comfortable with your main database(s), you can set up an *External Search Index* by deploying a product dedicated to the special kind of search that you need and feeding it updates from your main database.

Historical Data, Data Archiving



It is common to store the history of sales in a database. However, once a month or two has passed, it is very unlikely that the historical records will ever be edited. And though they are queried on very rare occasions, like audits, they still slow down your database. Some businesses offload any data older than a couple of months to a cheaper [archive storage](#) which does not allow changes to the data and has limited query capabilities in order to keep the main datasets small and fast.

Database Cache, Cache-Aside



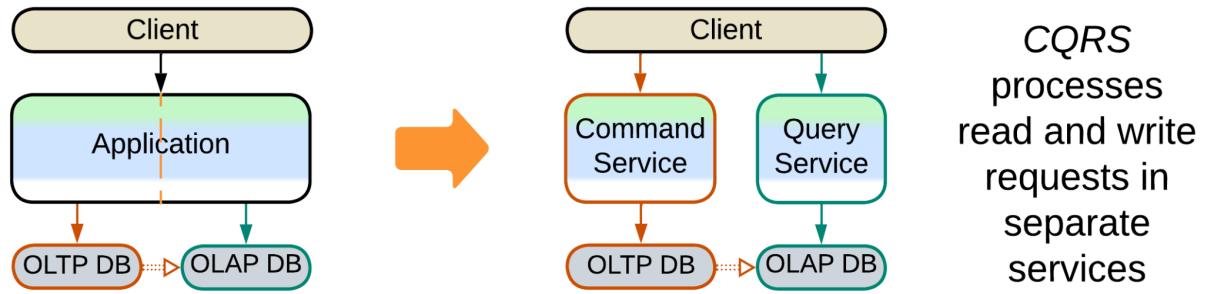
Database queries are resource-heavy while databases scale only to a limited extent. That means that a highly loaded system benefits from bypassing its main database with as many queries as possible, that is usually achieved by storing recent queries and their results in an in-memory database ([Cache-Aside](#)). Each incoming query is first looked for in the fast cache, and if it is found then you are lucky to get the result immediately without having to consult the main database.

Keeping the cache consistent with the main database is the hard part. There are quite a few strategies (some of them treat the [cache as a Proxy](#) for the database): [write-through](#), [write-behind](#), [write-around](#) and [refresh-ahead](#).

Evolutions

Polyglot Persistence with derived storage can often be made subject to CQRS:

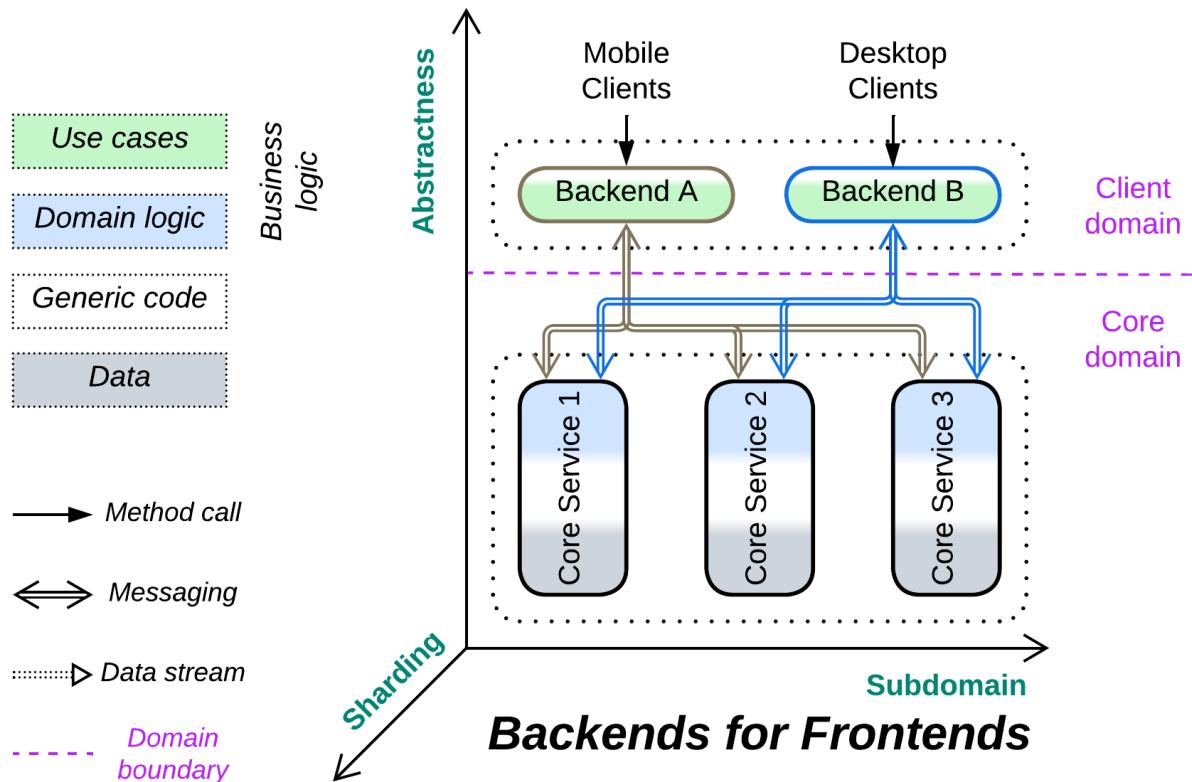
- The service that uses the read and write databases is [split into separate read and write services](#).



Summary

Polyglot Persistence employs several specialized databases to improve performance, often at the cost of eventual data consistency or implementing transactions in the application.

Backends for Frontends (BFF)



Hire a local guide. Dedicate a service for every kind of client.

Known as: Backends for Frontends (BFF), [Layered Microservice Architecture](#).

Aspects:

- [Proxy](#),
- [Orchestrator](#).

Variants:

Applicable to:

- Proxies,
- Orchestrators,
- Proxy + Orchestrator pairs,
- API Gateways,
- Event Mediators.

Structure: A layer of integration services over a shared layer of core services.

Type: Extension, derived from [Orchestrator](#) and/or [Proxy](#).

Benefits	Drawbacks
----------	-----------

Clients become independent in protocols, workflows, and, to an extent, forces

A specialized team and technology per client may be employed

The multiple *Orchestrators* are smaller and more cohesive than the original universal one

References: The [original article](#), a [smaller one](#) from Microsoft and an [excerpt](#) from [MP]. Here are [reference diagrams](#) from WSO2 (notice multiple *Microgateway* + *Integration Microservice* pairs).

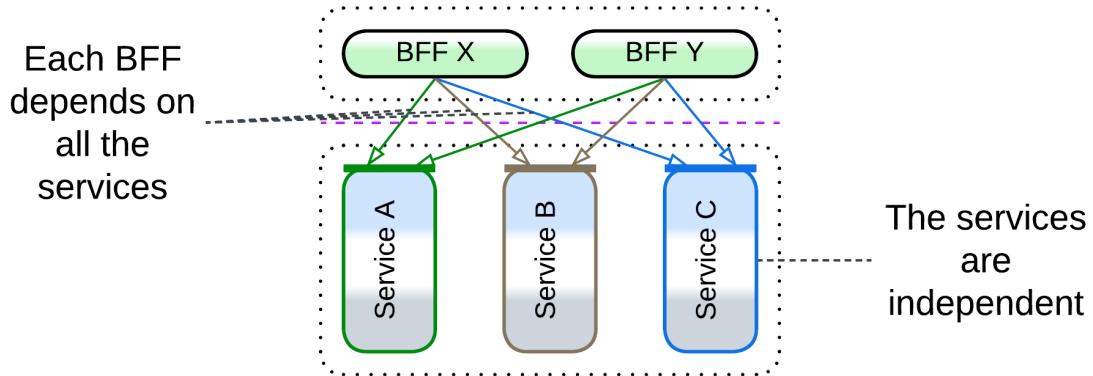
If some aspect(s) of serving our system's clients strongly vary by client type (e.g. OLAP vs OLTP, user vs admin, buyer vs seller vs customer support), it makes sense to use a dedicated component (the titular *Backend for Frontend* or *BFF*) per client type to encapsulate the variation. Protocol variations call for multiple [Proxies](#), workflow variations – for several [Orchestrators](#), both coming together – for [API Gateways](#) or [Proxy + Orchestrator](#) pairs. It is even possible to vary the *BFF*'s programming language on a per client basis. The drawback is that once the clients get their dedicated *BFFs* it becomes hard to share a common functionality between them, unless you are willing to add yet another new utility [service](#) or [layer](#) that can be used by each of them (and that will strongly smell of [SOA](#)).

Performance

As the multiple *Orchestrators* of *BFF* don't intercommunicate, the pattern's performance is identical to that of an [Orchestrator](#): it also slows down request processing in the general case but allows for several [specific optimizations](#), including direct communication channels between orchestrated [services](#).

Dependencies

Each *BFF* depends on all the services it uses (usually every service in the system). The services themselves are likely to be independent, as is common in [orchestrated systems](#).



Applicability

Backends for Frontends are good for:

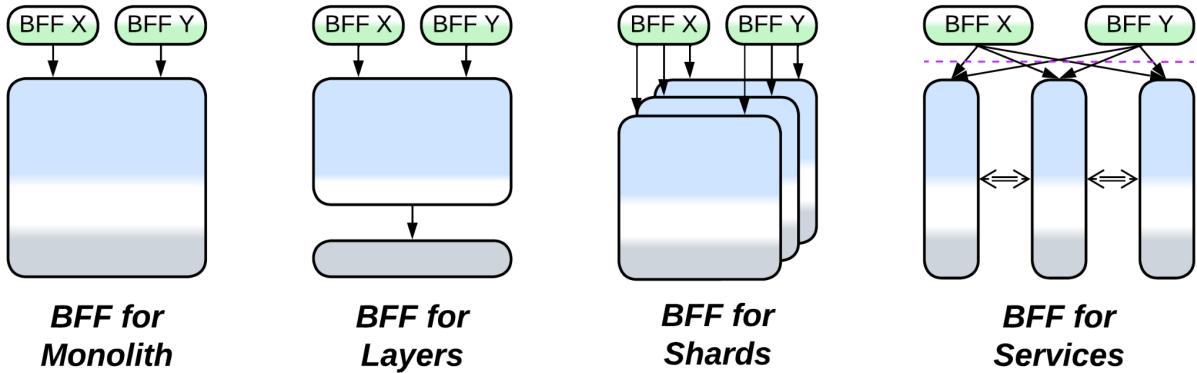
- *Multiple client protocols*. Deploying a [Gateway](#) per protocol hides the variation from the underlying system.
- *Multiple UIs*. When you have one team per UI, each of them may [want to have](#) an API which they feel comfortable with.
- *Drastically different workflows*. Let each client-facing development team own a component and choose the best fitting technologies and practices.

Backends for Frontends should be avoided when:

- *The clients are mostly similar*. It is hard to share code and functionality between *BFFs*. If the clients have much in common, the shared aspects either find their place in a shared monolithic layer (e.g. multiple client protocols call for multiple [Gateways](#) but a shared [Orchestrator](#)) or are duplicated. *BFF* may not be the best choice – use

OOD (conditions, factories, strategies, inheritance) instead to handle the clients' differences within a single codebase.

Relations



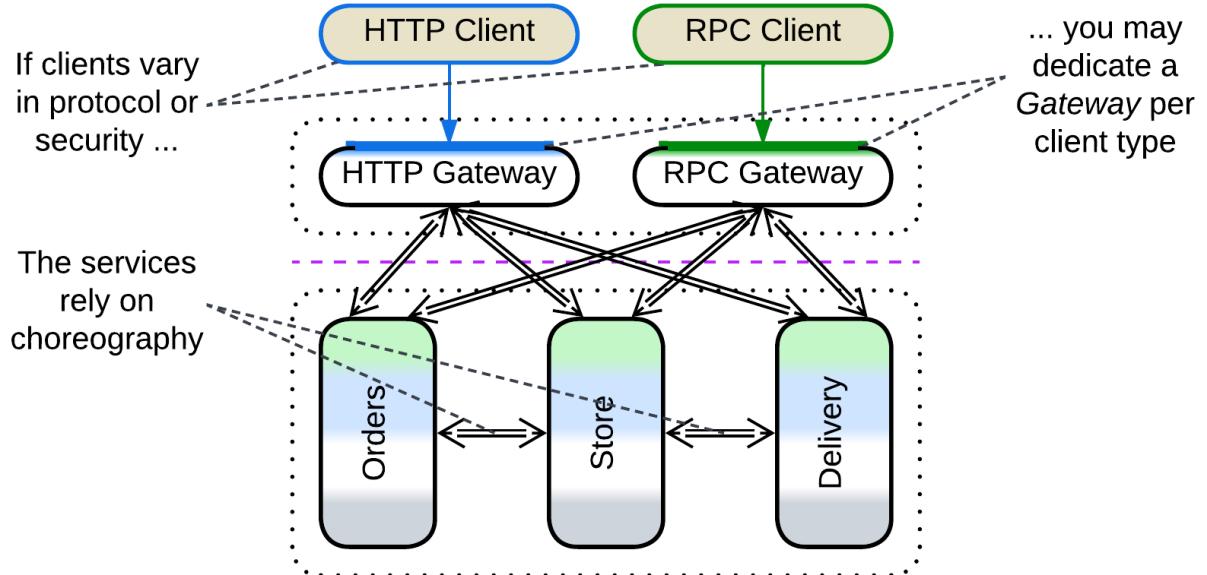
Backends for Frontends:

- Extends [Services](#) or rarely [Monolith](#), [Layers](#), or [Shards](#).
- Is derived from a client-facing extension metapattern: [Gateway](#), [Orchestrator](#), [API Gateway](#), or [Event Mediator](#).

Variants

Backends for Frontends vary in the kind of component that gets dedicated to each client:

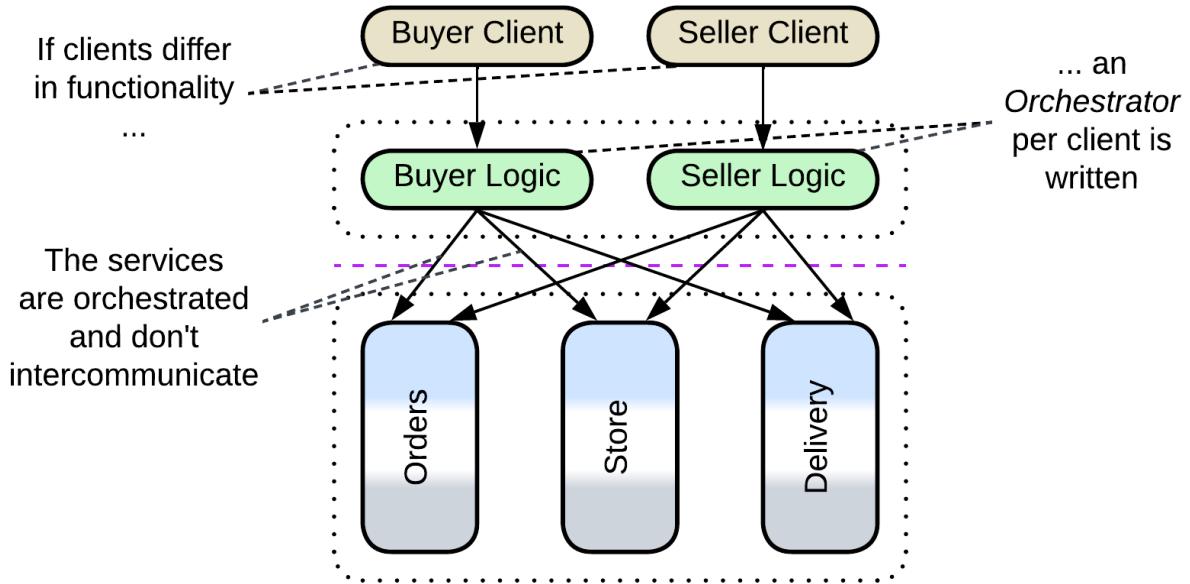
Proxies



Dedicating a [Gateway](#) per client is useful when the clients differ in the mode of access to the system (protocols / encryption / authorization) but not in workflows.

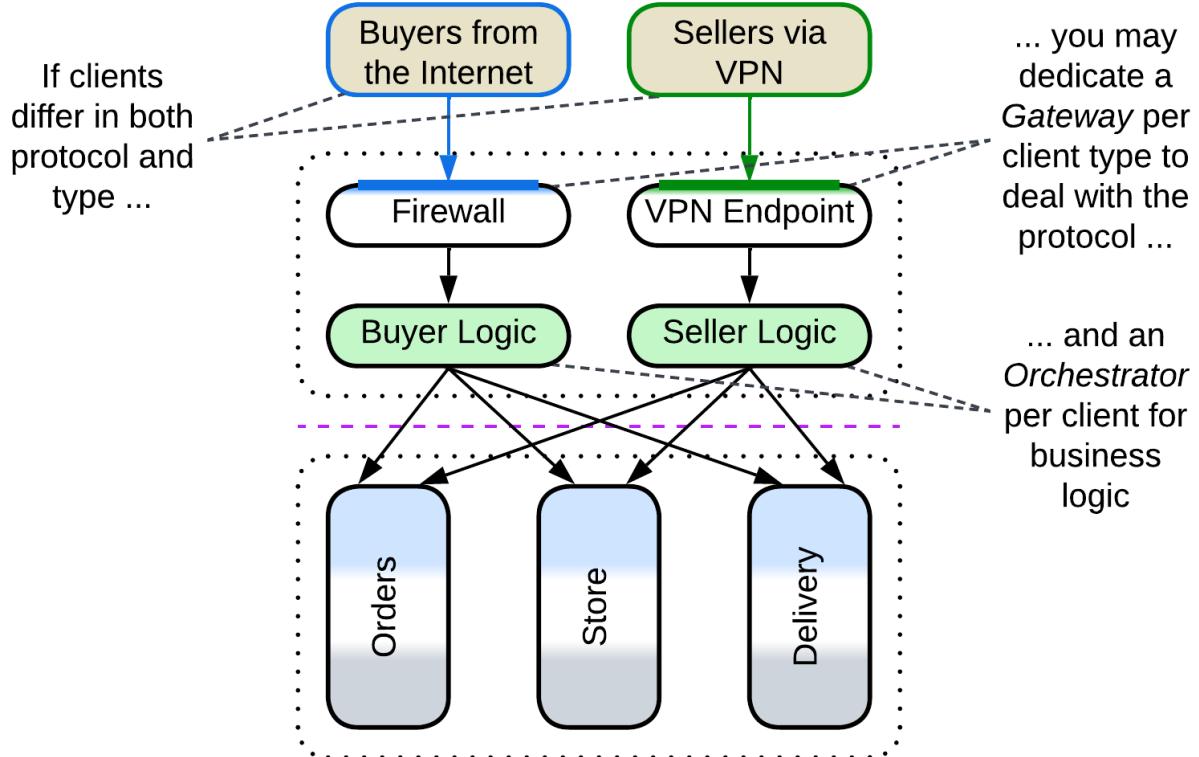
Multiple *Adapters* match the literal meaning of *Backends for Frontends* – each UI team ([backend](#), [mobile](#), [desktop](#); or [end-device-specific](#) teams) gets some code on the backend side to adapt the system's API to its needs by building a new, probably more high-level specialized API on top of it.

Orchestrators



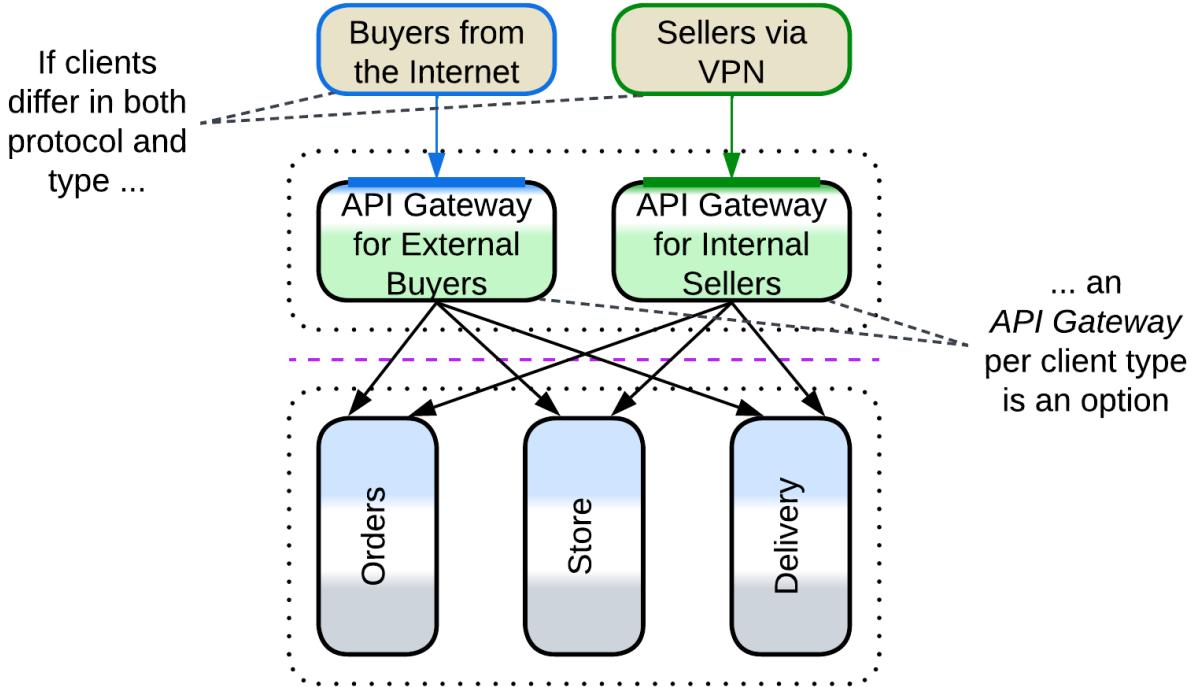
An [Orchestrator](#) per client makes sense if the clients use the system in completely unrelated ways, e.g. a shop's customers have little to share with its administrators.

Proxy + Orchestrator pairs



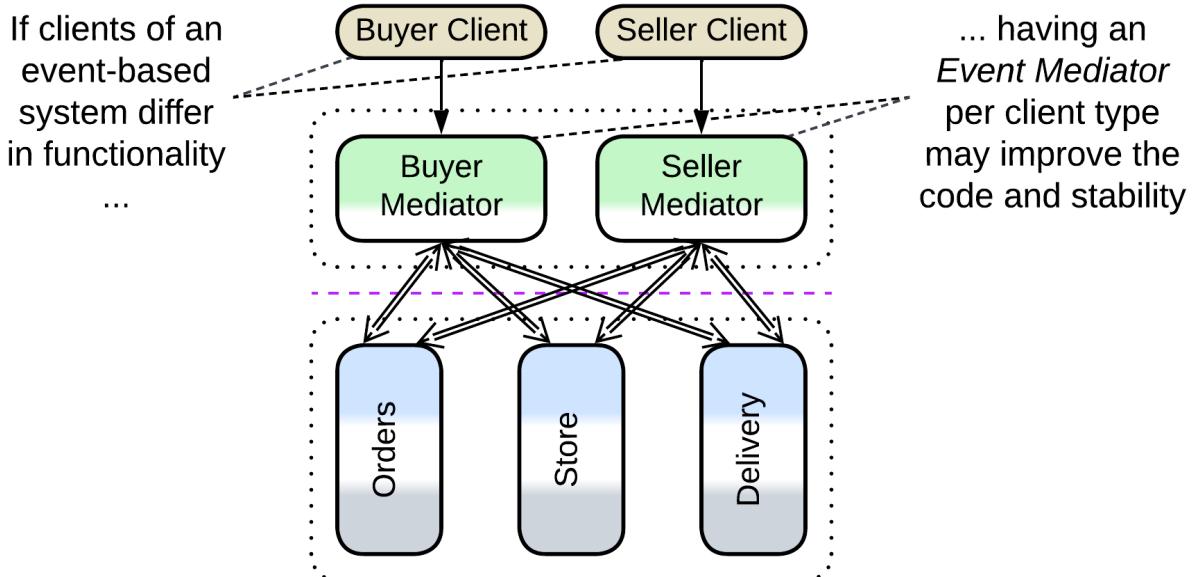
Clients vary in both access mode (protocol) and workflow. [Orchestrators](#) or [Proxies](#) may be reused if some kinds of clients share only protocol or application logic.

API Gateways



Clients vary in access mode (protocol) and workflow and there is a third-party [API Gateway](#) framework which seems to fit your requirements off the shelf.

Event Mediators



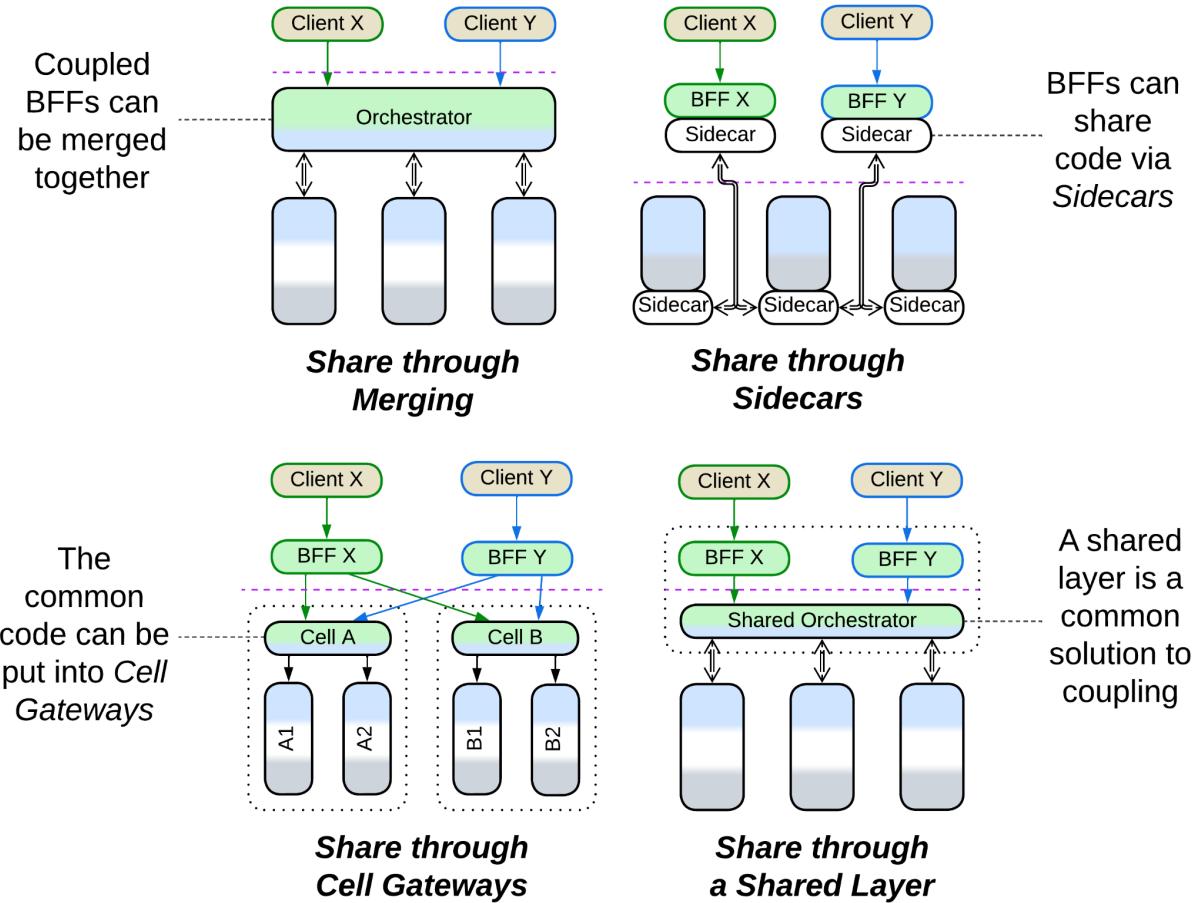
[FSA] mentions that multiple [Event Mediators](#) may be deployed in [Event-Driven Architecture](#) to split the codebase and improve stability.

Evolutions

BFF-specific evolutions aim at sharing logic between the BFFs:

- The BFFs can be merged into a single [Orchestrator](#) if their functionality becomes mostly identical.

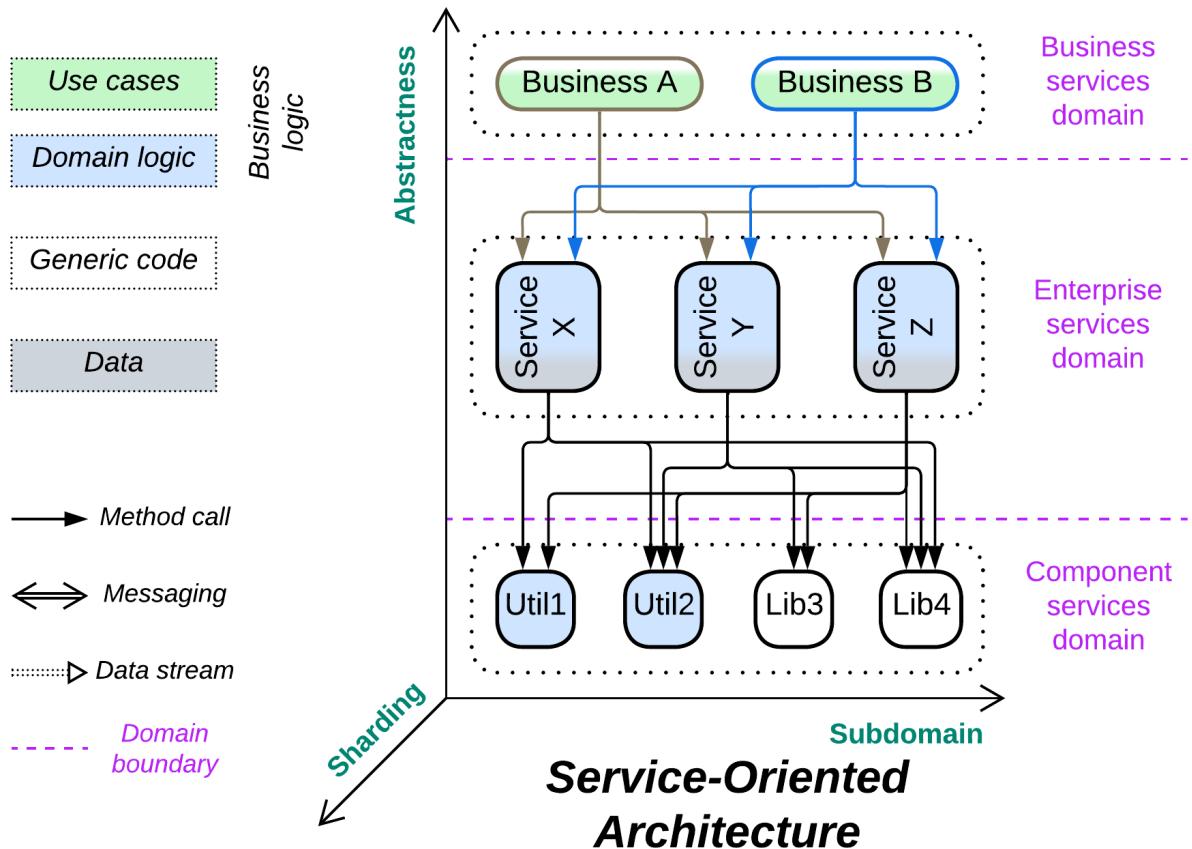
- A shared *orchestration layer* with common functionality may be added for use by the BFFs.
- A layer of *Integration Services* under the BFFs simplifies them by providing shared high-level APIs for the resulting *Cells*.
- *Sidecars [DDS]* (of *Service Mesh*) are a way to share libraries among the BFFs.



Summary

Backends for Frontends assigns a *Proxy* and/or an *Orchestrator* per each kind of a system's client to encapsulate client-specific use cases and protocols. The drawback is that there is no decent way for sharing functionality between the BFFs.

Service-Oriented Architecture (SOA)



The whole is equal to the sum of the parts. Distributed [Object-Oriented Design](#).

Known as: Service-Oriented Architecture (SOA), [Segmented Architecture](#).

Variants:

- Distributed Monolith,
- Enterprise SOA,
- [Domain-Oriented Microservice Architecture](#) (DOMA),
- (misapplied) Automotive SOA,
- [Nanoservices](#).

Structure: Usually three layers of services where each service can access any other service in its own or lower layers.

Type: Main, derived from [Layers](#).

Benefits	Drawbacks
Supports huge codebases	Very hard to debug
Multiple development teams and technologies	Hard to test as there are many dependencies
Forces may vary between the components	Very poor latency
Deployment to dedicated hardware	Very high DevOps complexity
Fine-grained scaling	The teams are highly interdependent

References: [FSA] has a chapter on Orchestration-Driven (Enterprise) Service-Oriented Architecture. [MP] mentions Distributed Monolith. There is also much (though somewhat conflicting) content over the Web.

Service-Oriented Architecture looks like the application of modular or object-oriented design followed by distribution of the resulting components over a network. The system usually contains three (rarely four) *layers of services* where every service has access to all the services below it (and sometimes some within its own layer). The services stay small, but as their number grows it becomes hard to keep in mind all the API methods and contracts available which a high-level component might use. Another issue originates from the idea of reusable components – multiple applications, written for different clients with varied workflows, require the same service to behave in (subtly) different ways, either causing its API to bloat or else impairing its usability (which means that a new customized duplicate service will likely be added to the system). Use cases are slow because there is much interservice communication over the network. Teams are interdependent as any use case involves many services, each owned by a different team. Testability is poor because there are too many moving (and being independently updated!) parts. The foundational idea of service reuse failed in practice, but its child architecture, *SOA*, still survives in historical environments.

Even though *SOA* fell from grace and is rarely seen in modern projects, it may soon be resurrected by low-code and no-code frameworks for serverless systems (e.g. [Nanoservices](#)) – it has everything ready: code reuse, granular deployment, and elastic scaling.

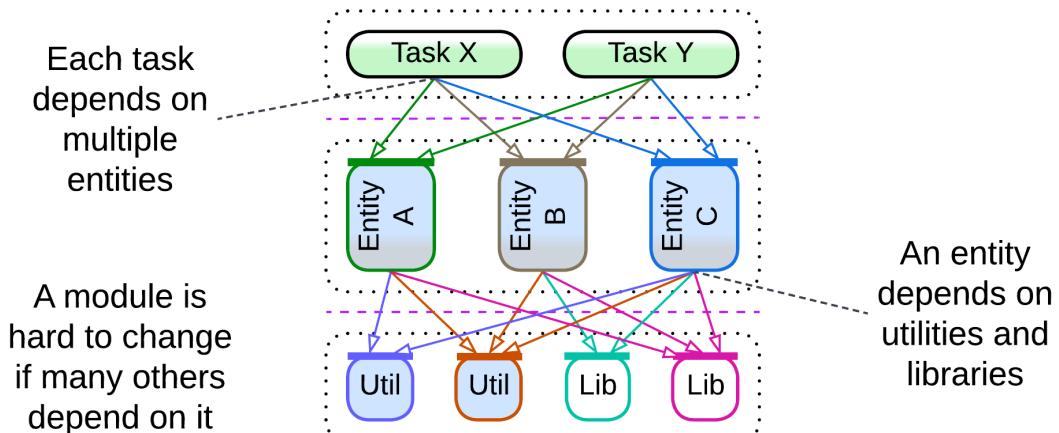
Performance

SOA is remarkable for its poor latency which results from extensive communication between its distributed components. There is hardly any way to help that as processing a request usually involves many services from all the layers.

Nevertheless, the pattern allows for good throughput as its stateless components can be scaled individually, leaving the system's scalability to be limited only by its databases, [Middleware](#) ... and funding.

Dependencies

Each service of each layer depends on everything it uses. As a result, development of a low-level (utility) component may be paralyzed because too many services already use it, thus no changes are welcome. Hence, the team writes a new version of their utility as a new service, which defeats the very idea of component reuse that *SOA* was based on.



Applicability

Service-Oriented Architecture is useful in:

- Huge projects. Many teams can be employed, each handling a moderate amount of code. However, dependencies between the teams and the combined length of the APIs in the system may [stall the development](#) anyway.
- A system of specialized hardware devices. If there is a lot of different hardware interacting in complex ways, the system may naturally fit the description of SOA. Don't fight this kind of [Conway's law](#).

Service-Oriented Architecture hurts:

- Fast-paced projects. Any feature requires coordination of multiple teams, which is hard to achieve in practice.
- Latency-sensitive domains. Over-distribution means too much messaging causing too high latency.
- High availability systems. Components may fail. A failure of a lower-level component is going to stall a large part of the system because every low-level component is used by many higher-level services.
- Safety-critical systems with frequent updates. SOA is hard to test comprehensively. Either all the components must be certified with a strict standard and an exhaustive test suite or any single component update requires re-testing the entire system.

Relations

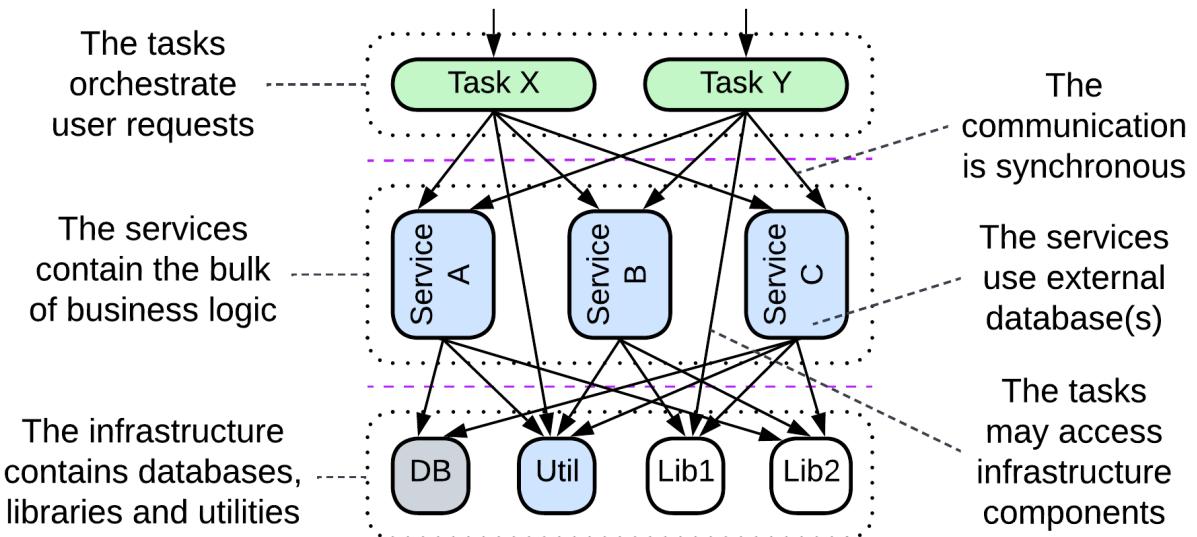
Service-Oriented Architecture:

- Is a stack of [Layers](#) each of which is divided into [Services](#).
- Is often extended with an [Enterprise Service Bus](#) (a kind of [orchestrating Middleware](#)) and one or more [Shared Databases](#).

Variants

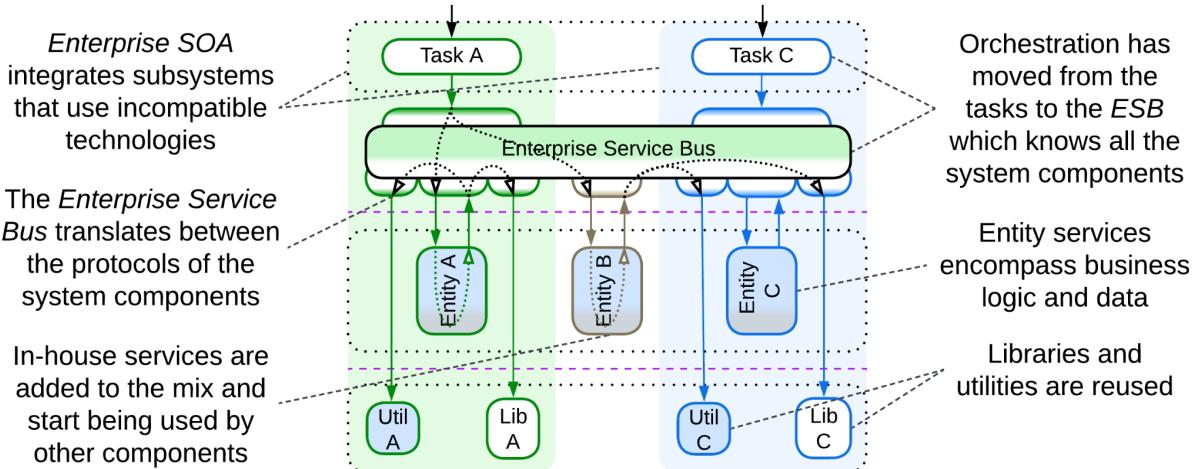
This architecture was hyped at the time when enterprises were expanding by acquiring smaller companies and conjoining their IT systems. The resulting merged systems were still heterogeneous and the development experience unpleasant, which inclined popular opinion towards the then novel notion of [Microservices](#). As nearly everybody has turned from merging existing systems to failing to apply Microservices in practice, the chance to find a pure greenfield SOA project in the wild is quite low. Many systems which are marketed as SOA are strongly modified:

Distributed Monolith



If a [Monolith](#) gets too complex and resource-hungry, the most simple & stupid way out of the trouble is to deploy each of its component modules to separate, dedicated hardware. The resulting services still communicate synchronously and are subject to domino effect on failure. Such an architecture may be seen as a (hopefully) intermediate structure in transition to more independent and stable event-driven [Services](#) (or [Cells](#)).

Enterprise SOA



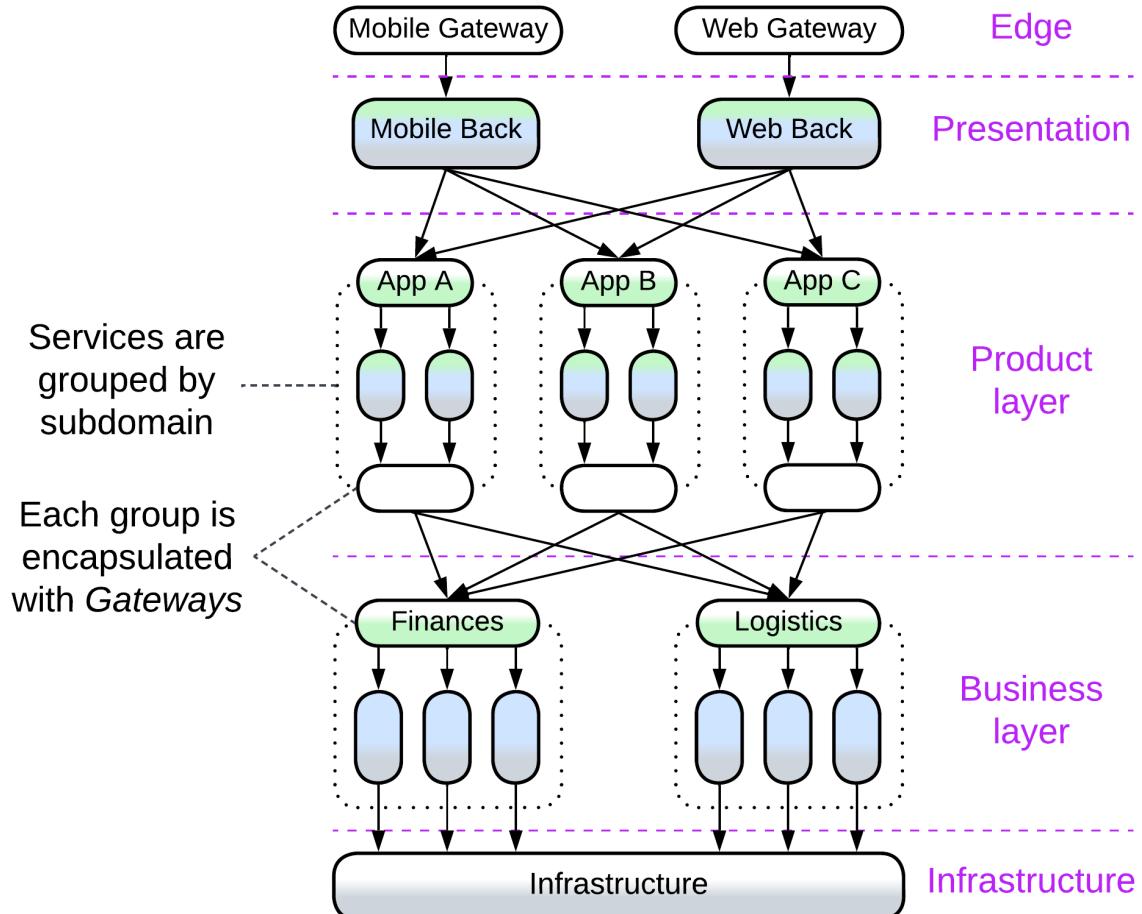
Multiple systems of [Services](#), each featuring an [API Gateway](#) and sometimes a [Shared Database](#), are integrated, resulting in new cross-connections. Much of the orchestration logic is removed from the [API Gateways](#) and reimplemented in an [orchestrating Middleware](#) called [Enterprise Service Bus \(ESB\)](#). This option allows for fast and only moderately intrusive integration (as no changes to the services, which implement the mass of the business logic, are required), but the single [orchestrating](#) component (*ESB*) often becomes the bottleneck for future development of the system due to its size and complexity. It is likely that if the orchestration were encapsulated in the individual [API Gateways](#), the system would be easier to deal with (making what is now [marketed by WSO2](#) as [Cell-Based Architecture](#)).

The layers of SOA are:

- *Business Process (Task)* – the definitions of use cases for a single business department, similar to the *API Gateways* layer of [BFF](#).

- *Services (Enterprise, Entity)* – the implementation of the business logic of a subdomain, to be used by the tasks.
- *Components (Application & Infrastructure, Utility)* – external libraries and in-house utilities that are designed for shared use by the services layer.

Domain-Oriented Microservice Architecture (DOMA)

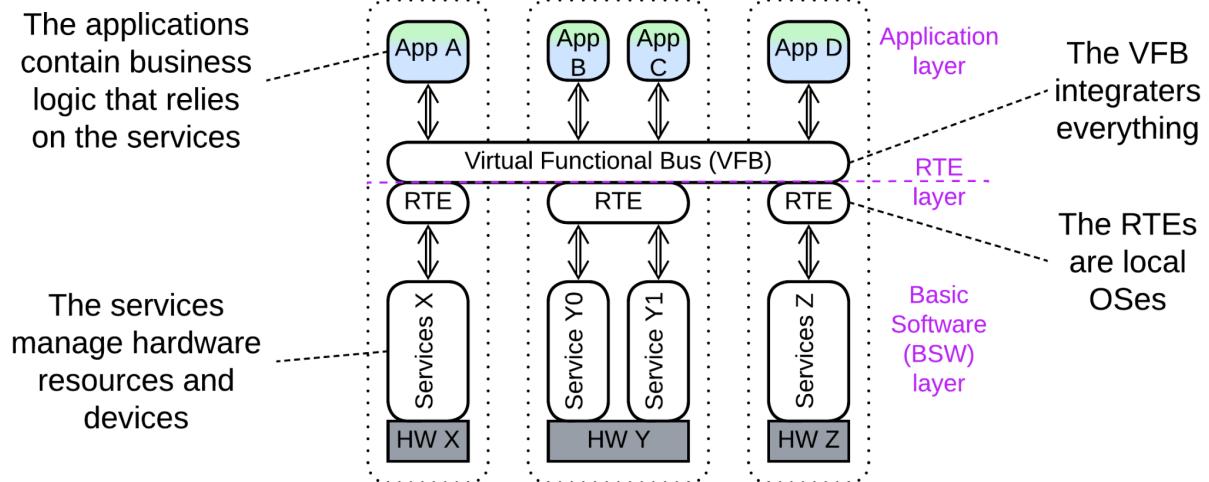


A huge business may build SOA of [Cells](#) (called *Domains*) instead of plain [Services](#). That greatly simplifies:

- administration (by reducing the number of components at the system level – Uber [packed](#) 2200 [Microservices](#) into 70 *Domains*),
- refactoring of individual subsystems (which are isolated behind [Cell Gateways](#)),
- development of business logic (as programmers need to learn much fewer interfaces of components they rely on).

Uber's DOMA also [makes heavy use](#) of [Plugins](#) which programmers from a client service team develop for the services they rely on. That allows for cross-*Domain* customization (injection of business logic from another *Cell*) of a service's behavior without making (slow) interservice calls.

(misapplied) Automotive SOA



Automotive architectures are marketed as *SOA*, but the old [AUTOSAR Classic](#) looks more like [Microkernel](#) (which indeed is similar to a 2-layered *SOA* with an [ESB](#)) while the newer system diagrams resemble a [Hierarchy](#). Therefore, they are addressed in the corresponding chapters.

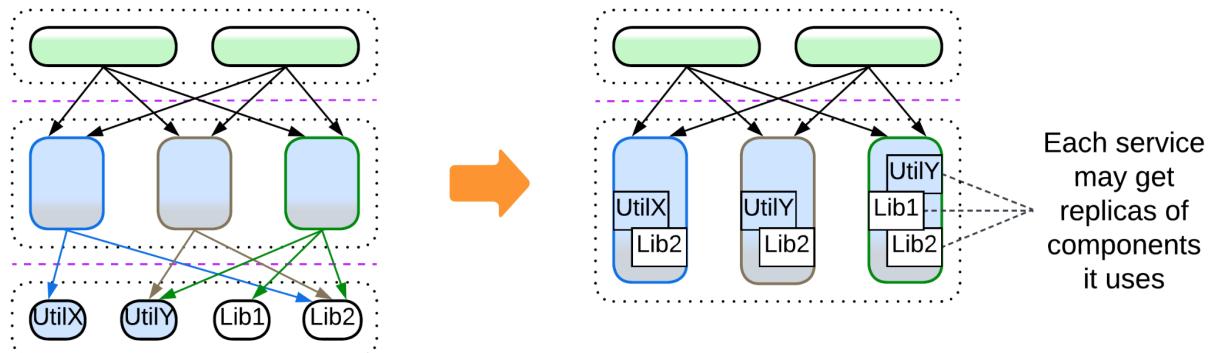
Nanoservices

It seems that some proponents of [Nanoservices take them](#) for a novel version of *SOA* – with the old good promise of reusable components. However, as that promise was failing miserably ever since the ancient days of OOP, it is no surprise that [in practice](#) *Nanoservices* are used instead to build [Pipelines](#) with little to no reuse.

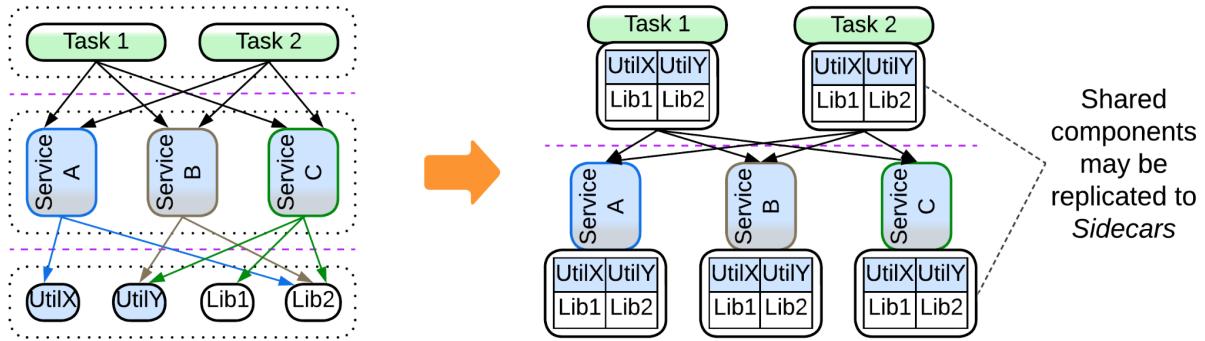
Evolutions

SOA suffers from excessive reuse and fragmentation. To fix that, first and foremost, each service of the *components (utility)* layer should be duplicated:

- Into every service that uses it, giving the developers who write the business logic full control of all the code that they use. Now they have several projects to support on their own (instead of asking other teams to make changes to their components).



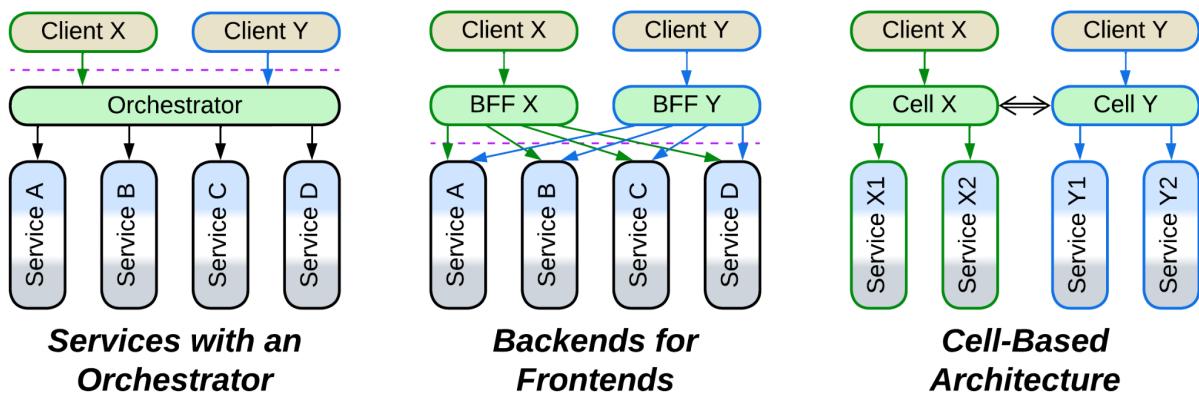
- Or into [Sidecars \[DDS\]](#) if you employ a [Service Mesh](#), resulting in much fewer network hops (thus lower latency) in request processing, but retaining the inter-team dependencies.



That removes the largest and most obvious part of the fragmentation, making the [ESB](#) (if you use one) [orchestrate](#) only the *entity* layer.

Afterwards you may deal with the remaining orchestration. The idea is to move the orchestration logic from the *ESB* to an explicit *layer of Orchestrators*:

- Either a monolithic *Orchestrator* over all the services.
- Or [Backends for Frontends](#) with an *Orchestrator* per client type (department of an enterprise) if each client uses most of the services.
- Or go for [Cells](#) with an *Orchestrator* per subdomain if your clients are subdomain-bound.
- Or a combination of the above.



Still another step is unbundling the [Middleware](#), which supports multiple protocols via [Adapters](#):

- If you have a [Service Mesh](#), an Adapter may be put to a [Sidecar](#) [[DDS](#)].
- Otherwise there is an option of a [hierarchical Middleware](#) (*Bus of Buses*) if closely related components share protocols.

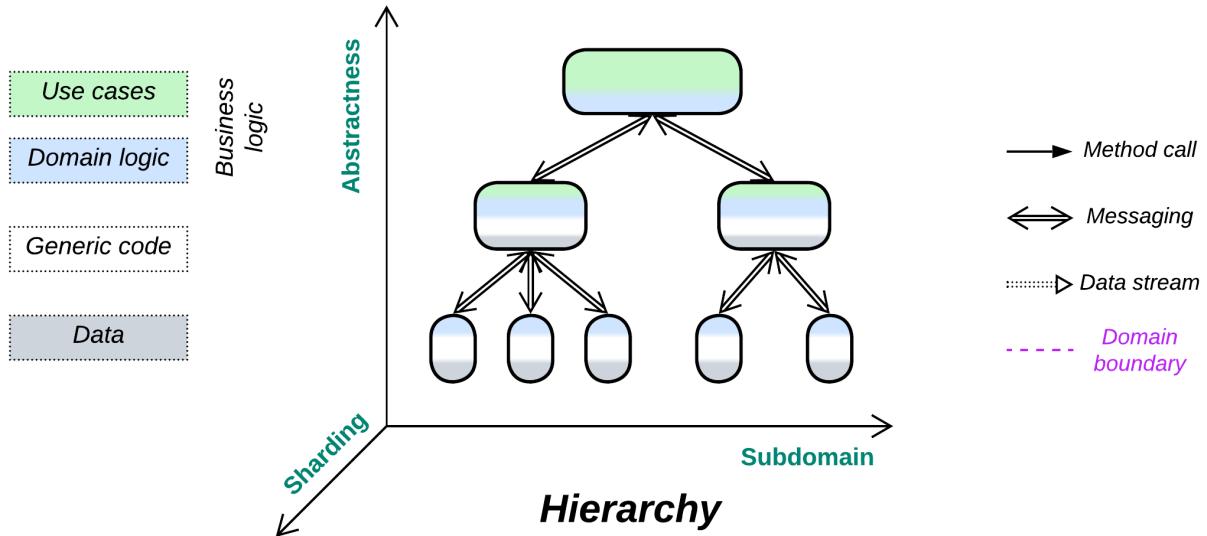
Still, the evolution of [Middleware](#) may not bring any real benefit except for removing the *ESB* altogether, which may not be that bad after all when it is not misused.

In any case, many of the evolutions will likely be very expensive, thus it makes sense to conduct some of them gradually via a kind of [strangler fig approach](#). Or let the architecture live and die as it is.

Summary

Service-Oriented Architecture divides each of the *integration*, *domain*, and *utility* layers into shared services. The extensive fragmentation and reuse degrade performance and speed of development. Nevertheless, huge projects are known to survive with this architecture.

Hierarchy



Command and conquer. Build a tree of responsibilities.

Variants:

By structure:

- Polymorphic children,
- Functionally distinct children.

By direction:

- Top-Down Hierarchy / Orchestrator of Orchestrators / [Presentation-Abstraction-Control](#) (PAC) [[POSA1](#), [POSA4](#)] / [Hierarchical Model-View-Controller](#) (HMVC),
- Bottom-Up Hierarchy / Bus of Buses / Network of Networks,
- In-Depth Hierarchy / [Cell-Based \(Microservice\) Architecture](#) (WSO2 version) / [Segmented Microservice Architecture](#) / Services of Services / Clusters of Services [[DEDS](#)].

Structure: A tree of components.

Type: Main or extension.

Benefits	Drawbacks
Very good in decoupling logic	Global use cases may be hard to debug
Supports multiple development teams and technologies	Poor latency for global use cases
Components may vary in qualities	Operational complexity
Low-level components are easy to replace	Slow start of the project

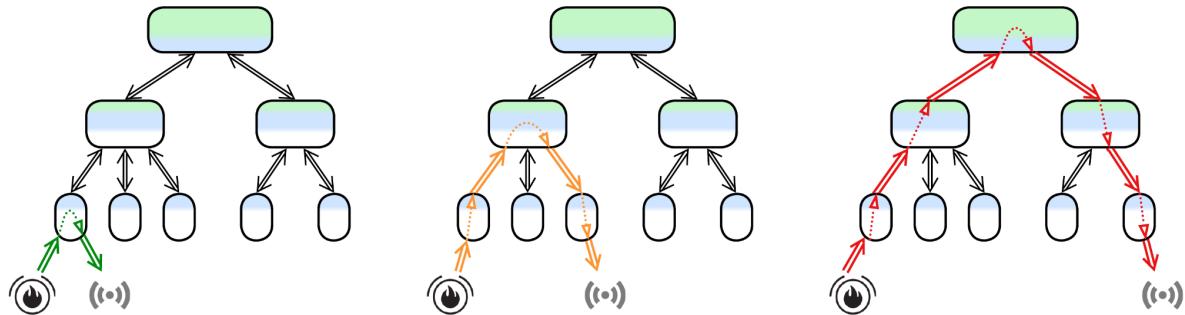
References: None good I know of.

Though not applicable to every domain, hierarchical decomposition is arguably the best way to distribute responsibilities between components. It limits the connections (thus the number of interfaces and contracts to keep in mind) of each component to its parent and a few children, allowing for the building of complex (and even complicated) systems in a simple way. The hierarchical structure is very flexible as it features [multiple layers of indirection](#) (and often polymorphism), which makes addition, replacement, or [stubbing/mock](#)ing of leaf components trivial. It is also quite fault-tolerant as individual subtrees operate independently.

This architecture is not ubiquitous because few domains are truly hierarchical. Its high fragmentation results in increased latency and poor debugging experience. Moreover, component interfaces should be designed beforehand and are hard to change.

Performance

No kind of distributed hierarchy is latency-friendly as many use cases involve several network hops. The fewer layers of the hierarchy are involved in a task, the better its performance.



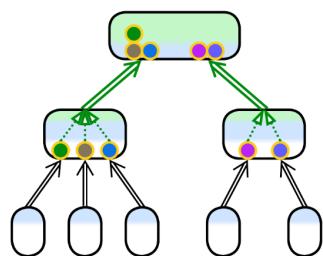
The fastest operation does not involve any interactions inside the hierarchy

Local actions in the hierarchy are slower

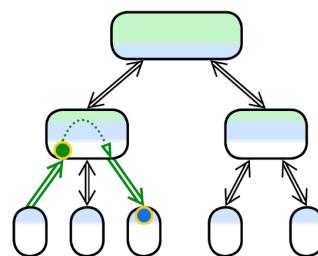
Actions that involve all the levels of the hierarchy are the slowest

Maintaining high throughput usually requires deploying multiple instances of the root component, which is not possible if it is stateful (in [control systems](#)) and the state cannot be split into [Shards](#). The following tricks may help unloading the root:

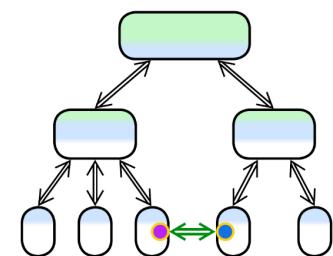
- *Aggregation* (first met in [Layers](#)): a node of a hierarchy collects reports from its children, aggregates them into a single package, and sends the aggregated data up to its parent. This greatly reduces traffic to the root in large [IIoT](#) networks.
- *Delegation* (resembles strategy injection and batching for [Layers](#)): a node should try to handle all the low-level details of communication with its children without consulting its parent node. For a [control system](#) that means that its mid-level nodes should implement control loops for the majority of incoming events. For a [processing system](#) that means that its mid-level nodes should expose coarse-grained interfaces to their parent(s) while translating each API method call into multiple calls to their child nodes.
- *Direct communication channels* (previously described for [Orchestrator](#)): if low-level nodes need to exchange data, their communication should not always go through the higher-level nodes. Instead, they may negotiate a direct link (open a socket) that bypasses the root of the hierarchy.



Mid-level nodes should aggregate the events they receive from child nodes



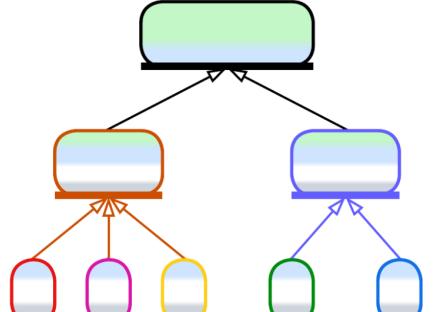
Mid-level nodes should act autonomously whenever possible



Low-level nodes can intercommunicate directly

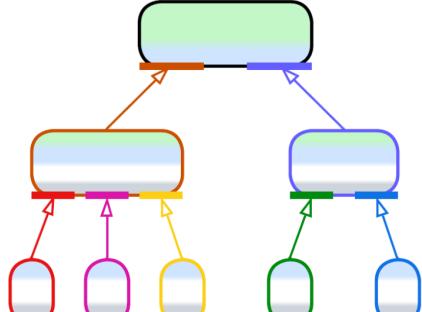
Dependencies

A parent node would usually define one (for polymorphic children) or more (otherwise) [SPIs](#) for its child nodes to implement. The interfaces reside on the parent side because low-level nodes tend to be less stable (new types of them are often added and old ones replaced) therefore we don't want our main business logic to depend on them.



Hierarchy with Polymorphism

In *Hierarchy* parent nodes tend to provide interfaces for their children



Hierarchy without Polymorphism

Applicability

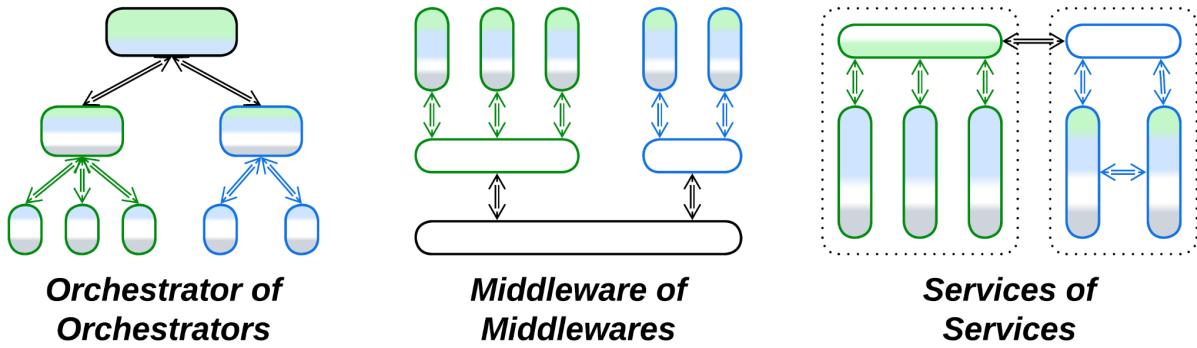
Hierarchy fits with:

- *Large and huge projects.* The natural division by both level of abstractness and subdomain allows for using smaller modules, ideally with intuitive interfaces. The APIs for each team to learn are limited to just a few which their component interacts with directly.
- *Systems of hardware devices.* Real-world [IIoT](#) systems may use a hierarchy of controllers to benefit from autonomous decision-making and data aggregation.
- *Customization.* The tree-like structure provides opportunities for easy customization. A medium-sized hierarchical system may integrate hundreds of leaf types.
- *Survivability.* A distributed hierarchy retains limited functionality even if several of its nodes fail.

Hierarchy fails with:

- *Cohesive domains.* Horizontal interactions (those between nodes that belong to the same layer) bloat interfaces as they have to pass through parent nodes.
- *Quick start.* Finding (and verifying) a good hierarchical domain model may be hard if at all possible. Debugging an initial implementation will not be easy.
- *Low latency.* System-wide scenarios involve many cross-component interactions which are slow in distributed systems.

Relations



Hierarchy:

- Can be applied to [Orchestrator](#), [Middleware](#) or [Services](#).

Variants by structure (may vary per node)

Hierarchy comes in various shapes as it is more of a design approach than a ready-to-use pattern:

Polymorphic children

All the managed child nodes expose the same interface and contract. This tends to simplify the implementation of the parent node and resembles inheritance of OOD.

Example: a fire alarm system may treat all of its fire sensors as identical devices, even though the real hardware comes from many manufacturers.

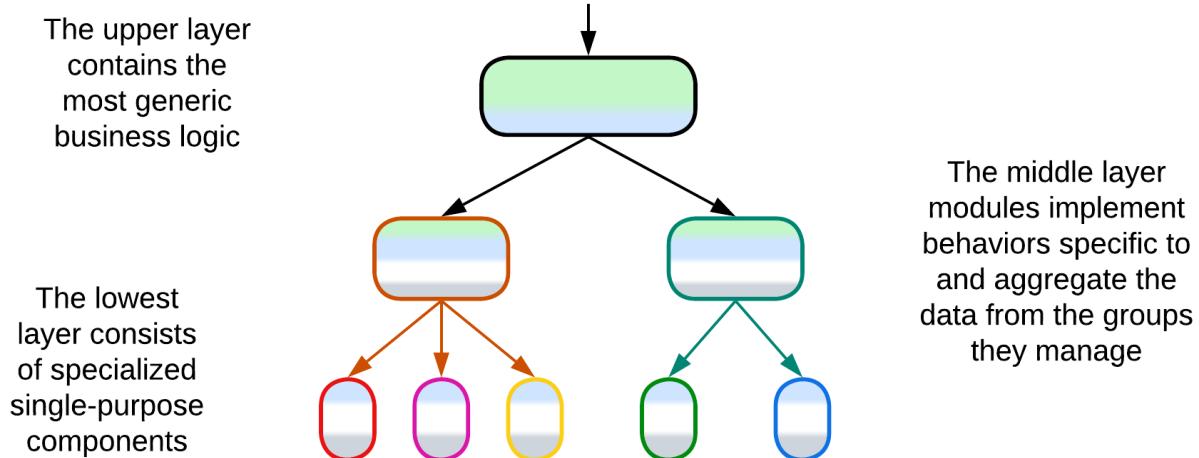
Functionally distinct children

The managing node is aware of several kinds of children that vary in their APIs and contracts, just like with composition in OOD.

Example: an intrusion alarm logic may need to discern between cat-affected IR sensors and mostly cat-proof glass break detectors.

Variants by direction

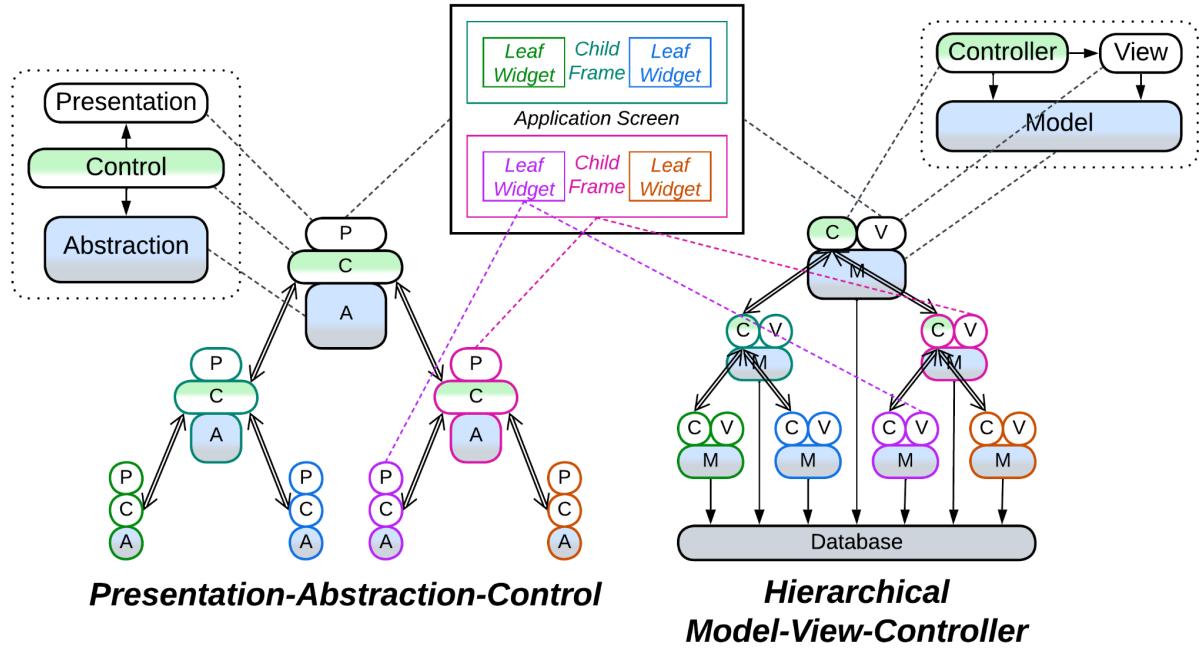
Top-Down Hierarchy, Orchestrator of Orchestrators, Presentation-Abstraction-Control (PAC), Hierarchical Model-View-Controller (HMVC)



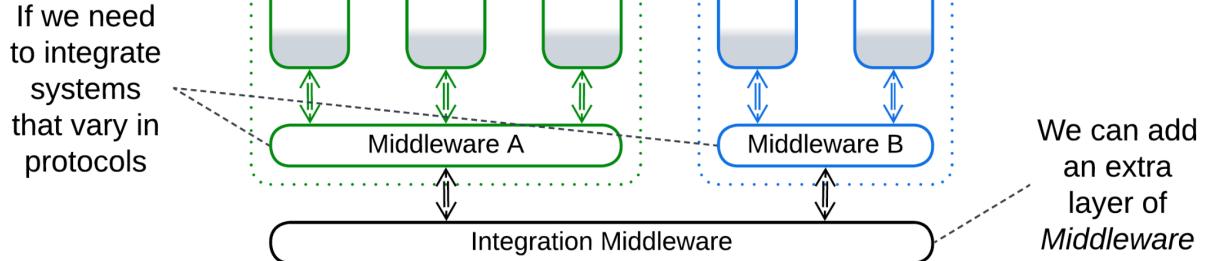
In the most common case *Hierarchy* is applied to business logic to build a layered system which grows from a single generic high-level root into a swarm of specialized low-level pieces. The most obvious applications are protocol parsers, decision trees, [IIoT](#) (e.g. a fire alarm system of a building), and [modern automotive](#) networks. A marketplace that allows for customized search and marketing algorithms within each category of its goods may also be powered by a hierarchy of category-specific services.

[Presentation-Abstraction-Control](#) (PAC) [[POSA1](#), [POSA4](#)] applies *Top-Down Hierarchy* to a user-facing application, providing each of the resulting *layered* nodes with its own widget (*presentation*) on the UI screen (which is the *presentation* of the root node). *Controls* are responsible for inter-node communication and integration logic, while domain logic and data reside in *abstractions*.

[Hierarchical Model-View-Controller](#) (HMVC) is similar, but its views access models directly, like in [MVC](#), and every model synchronizes with the global data. This pattern [was used](#) in [rich clients](#).



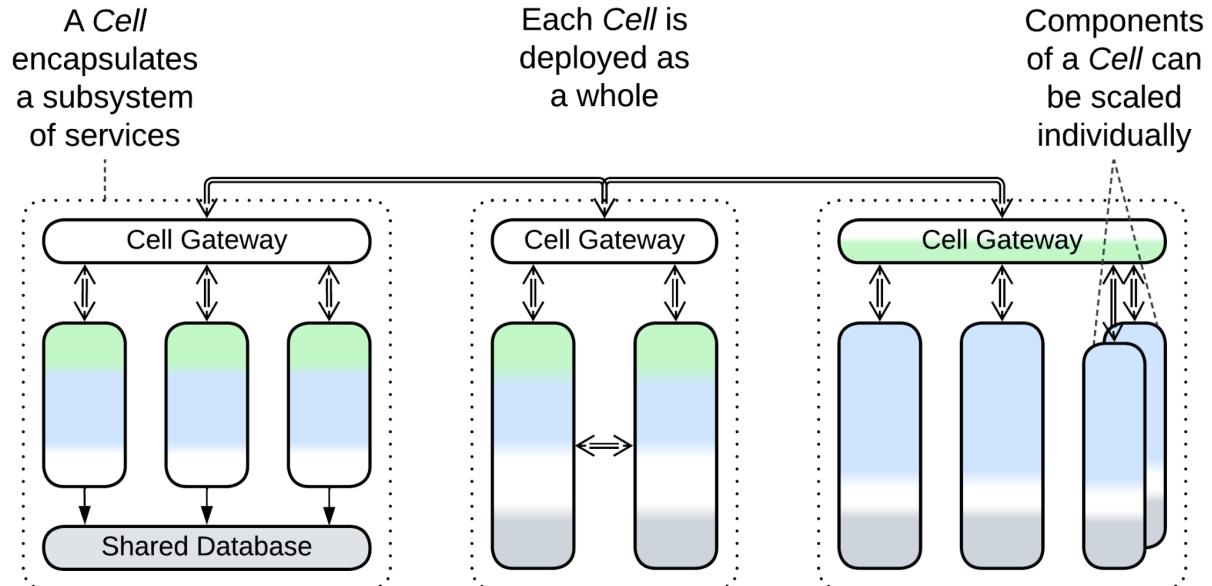
Bottom-Up Hierarchy, Bus of Buses, Network of Networks



Other cases require building a common base for intercommunication between several networks which vary in their protocols (and maybe even their hardware). The root of such a **Hierarchy** is a **Middleware** generic and powerful enough to cover the needs of all the specialized networks which it interconnects.

Example: [Automotive networks](#), integration of corporate networks, [the Internet](#).

In-Depth Hierarchy, Cell-Based (Microservice) Architecture (WSO2 version), Segmented Microservice Architecture, Services of Services, Clusters of Services



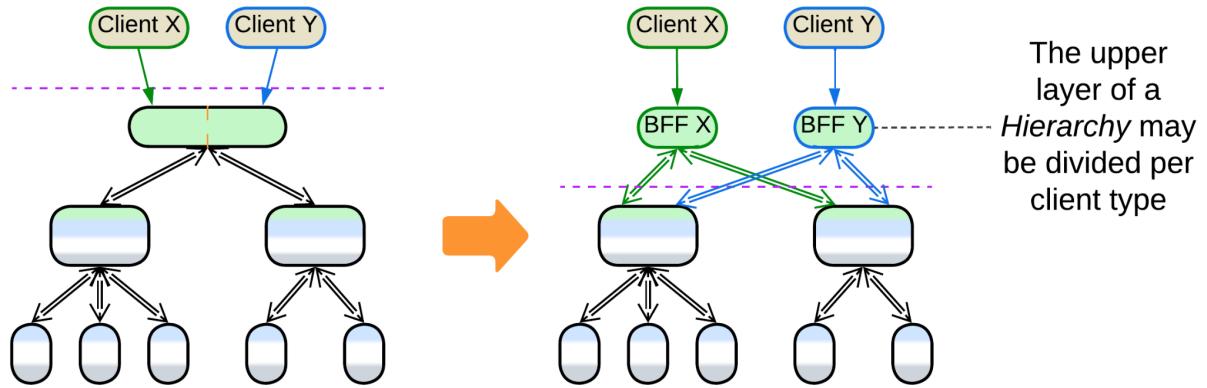
When several [services](#) in a system grow large, in some cases it is possible to divide each of them into [subservices](#). Each group of the resulting subservices (known as a [Cell](#), [Domain](#) or [Cluster \[DEDS\]](#)) usually implements a [bounded context](#) [[DDD](#)]. It is hidden behind its own [Cell Gateway](#) and may even use its own [Middleware](#). Subservices of a [Cell](#) may [share a database](#) and may be deployed as a single unit. This keeps the system's integration complexity (the length of its APIs and the number of deployable units) reasonable while still scaling development among many teams, each owning a service. If each instance of a [Cell](#) owns a [shard](#) of its database, the system [becomes more stable](#) as there is no single point of failure (except for the [Load Balancer](#) called [Cell Router](#)). Another benefit is that [Cells](#) can be deployed to regional data centers to improve locality for users of the system. However, that will likely cause data synchronization traffic between the data centers.

The [Cell-Based Architecture \(Segmented Microservice Architecture\)](#) may be seen as a combination of an [Orchestrator of Orchestrators](#) and a [Bus of Buses](#) where the subservices are leaf nodes of both [hierarchies](#) while the [API Gateways](#) of the [Cells](#) are their internal nodes.

Uber [compacted](#) 2200 [Microservices](#) into 70 [Cells](#) arranged in [SOA](#)-style [Layers](#) called [Domain-Oriented Microservice Architecture](#).

Evolutions

- The upper component of a [Top-Down Hierarchy](#) can be split into [Backends for Frontends](#).



Summary

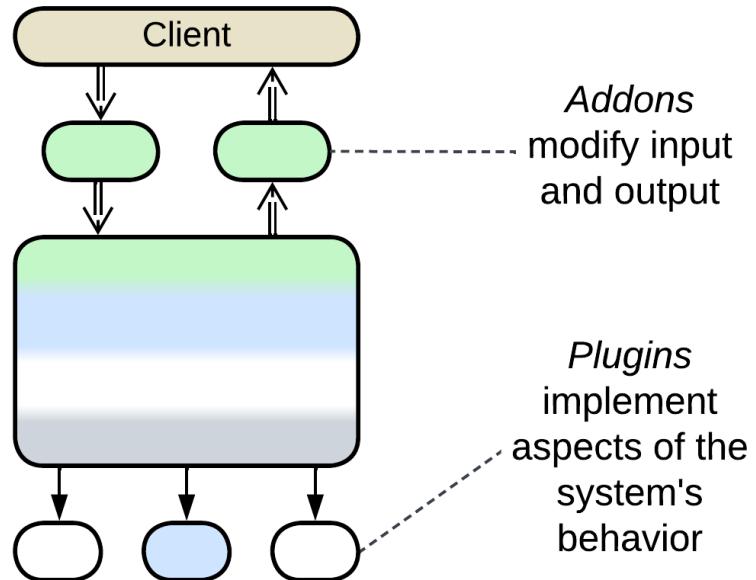
Hierarchy fits a project of any size as it evenly distributes complexity among the system's many components. However, it is not without drawbacks in performance, debuggability, and operational complexity. Moreover, very few domains allow for seamless application of this architecture.

Part 5. Implementation Metapatterns

There are patterns that describe implementation of components:

Plugins

Plugins allow for customization of the system's functionality

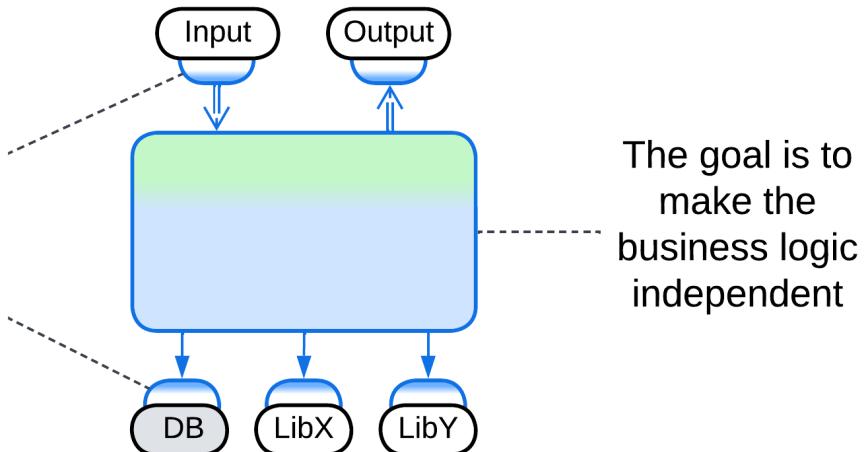


The *Plugins* pattern is about separating a system's main logic from the customizable details of its behavior. That allows for the same codebase to be used for multiple flavors or customers.

Includes: Plug-In Architecture, Addons, Strategy, Hooks.

Hexagonal Architecture

Hexagonal Architecture assigns an adapter to each external component

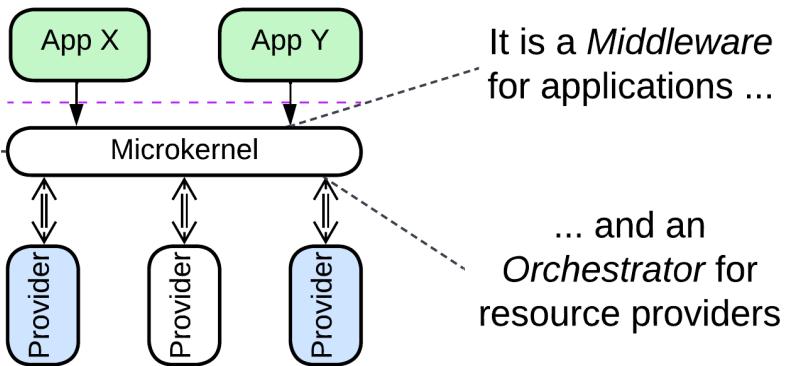


Hexagonal Architecture is a specialization of *Plugins* where every external dependency is isolated behind an *Adapter*, making it easy to update or replace third-party components.

Includes: Ports and Adapters, Onion Architecture, Clean Architecture; Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), Model-View-Controller (MVC), and Action-Domain-Responder (ADR).

Microkernel

The *microkernel* shares resources of providers among applications

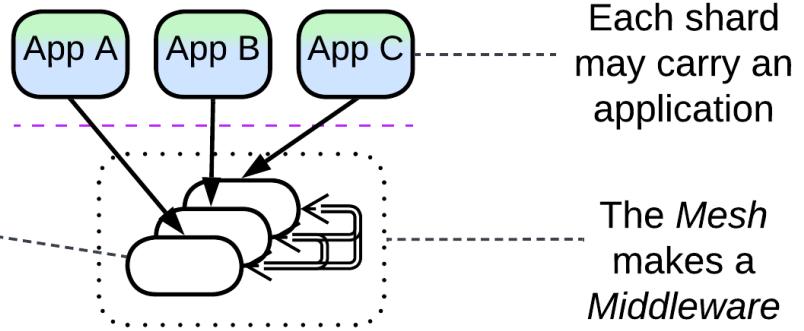


This is another derivation of *Plugins*, with a rudimentary *core component* which mediates between resource consumers (*applications*) and resource providers. The *Microkernel* is a *Middleware* to the *applications* and an *Orchestrator* to the *providers*.

Includes: operating system, software framework, virtualizer, distributed runtime, interpreter, configuration file, Saga Engine, AUTOSAR Classic Platform.

Mesh

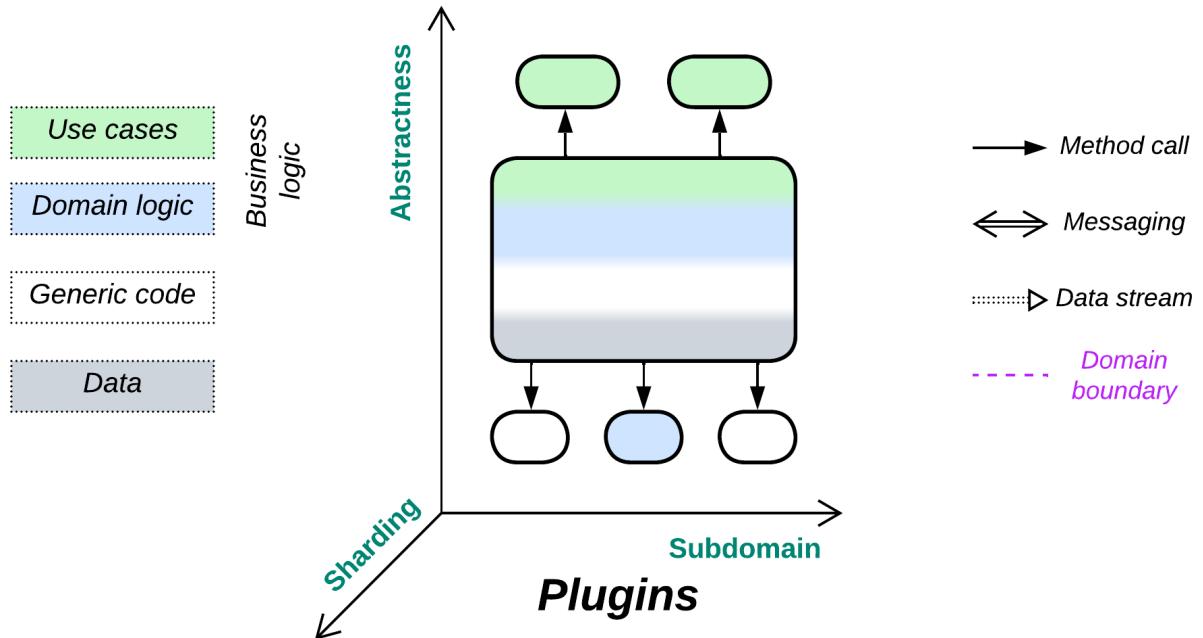
Mesh comprises interconnected *Shards*



A *Mesh* consists of intercommunicating shards, each of which may host an application. The shards coalesce into a fault-tolerant distributed *Middleware*.

Includes: grid; peer-to-peer networks, Leaf-Spine Architecture, Actors, Service Mesh, Space-Based Architecture.

Plugins



Overspecialize, and you breed in weakness. Customize the system through attachable modules.

Known as: Plug-In Architecture [[FSA](#)], (misapplied) Microkernel (Architecture) [[POSA1](#), [POSA4](#), [SAP](#), [FSA](#)], Plugin [[PEAA](#)], Addons, Strategy [[GoF](#), [POSA4](#)], Reflection [[POSA1](#), [POSA4](#)], Aspects, Hooks.

Variants: [Hexagonal Architecture](#) and [Microkernel](#) got dedicated chapters. Plugins have many variations.

Structure: A [Monolith](#) extended with one or more modules that customize its behavior.

Type: Implementation, extension.

Benefits	Drawbacks
Some aspects are easy to customize	Testability is poor (too many combinations)
The customized system is relatively light-weight	Performance is suboptimal
Platform-specific optimizations are supported	Designing good plugin APIs is hard
The custom pieces may be written in a different programming language or DSL	

References: [[SAP](#)] and [[FSA](#)] [mistakenly](#) call this pattern *Microkernel* and dedicate chapters to it.

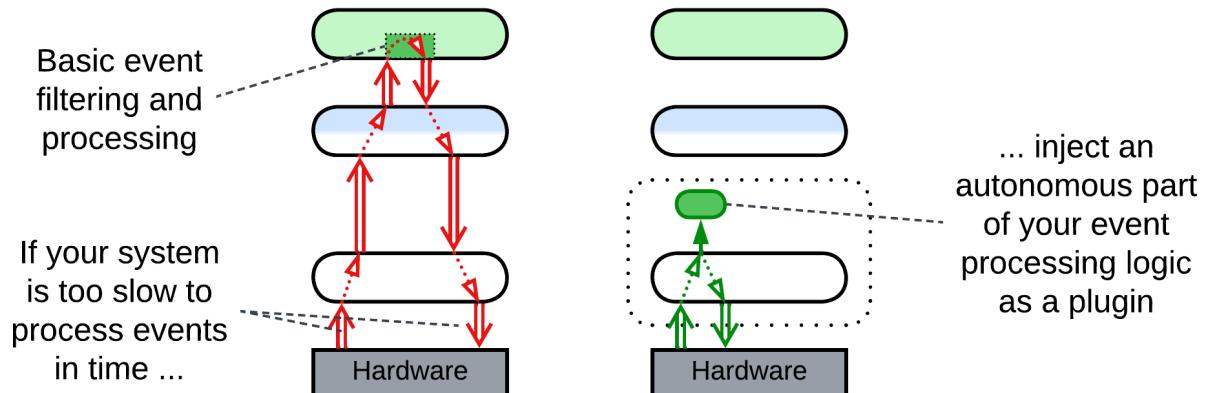
Most systems require some extent of customizability all the way from the basic codec selection by a video player to the screens full of tools and wizards unlocked once you upgrade your subscription plan. This is achieved by keeping the *core* functionality separate from its extensions, which are developed by either your team or external enthusiasts to modify behavior of the system. The cost of flexibility is paid in complexity of design – the need to predict which aspects should be customizable and which APIs are good for known (and unknown) uses by the extensions. Heavy communication between the core and *plugins* negatively impacts performance.

Performance

Using plugins usually degrades performance. The effect may be negligible for in-process plugins (such as strategies or codecs) but it becomes noticeable if inter-process communication or networking is involved.

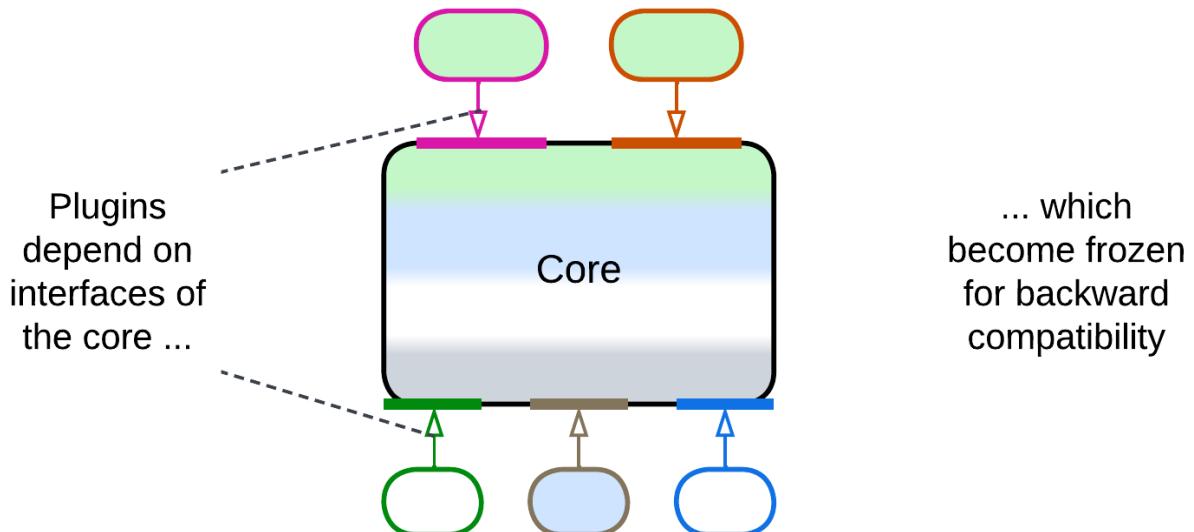
The only case for a plugin to improve performance of a system that I can think up is when a part of the client's business logic moves to a plugin of a lower layer of a system. That is similar to the [strategy injection optimization for Layers](#). Real-world examples include:

- The use of *stored procedures* in databases,
- HFT rules and price tables [uploaded](#) to a network card or FPGA,
- Customization of supplier services for varying needs of their client services in [Domain-Oriented Microservice Architecture](#).



Dependencies

Each *plugin* depends on the *core's API* (for *Addons*) or *SPI* (for *Plugins*) for the functionality it extends. That makes the APIs and SPIs nearly impossible to modify, only to extend, as there tend to be many plugins in the field, some of them out of active development, that rely on any given method of the already published interfaces.



Applicability

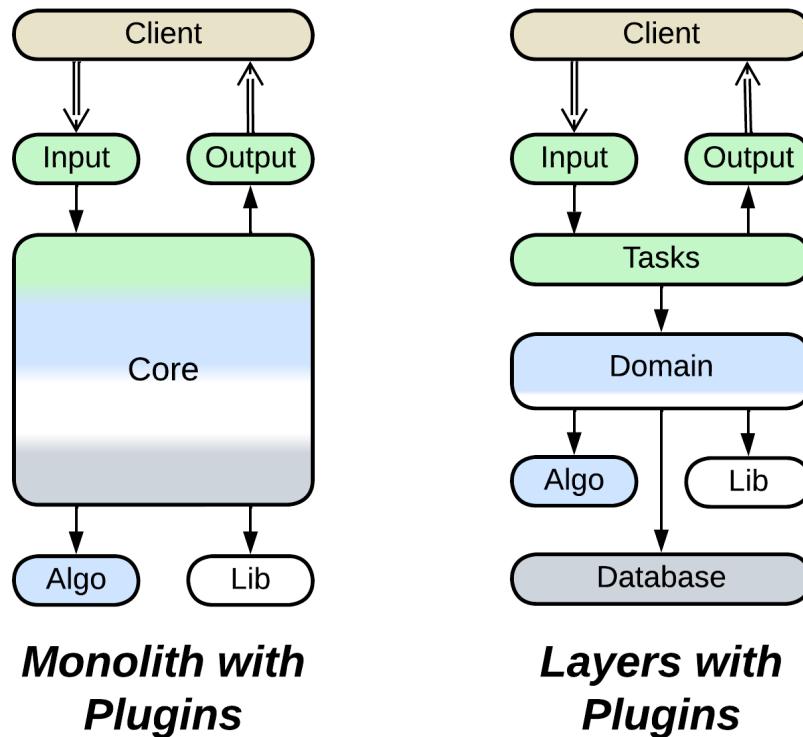
Plugins are good for:

- *Product lines*. The shared core functionality and some *plugins* are reused by many flavors of the product. Per-client customizations are largely built using existing plugins.
- *Frameworks*. A framework is a functional core to be customized by its users. When shipped with a set of plugins it becomes ready-to-use.
- *Platform-specific customizations*. *Plugins* allow both for native look and feel (e.g. desktop vs mobile vs console) and for the use of platform-specific hardware.

Plugins do not perform well in:

- *Highly optimized applications*. Any generic code tends to be inefficient. A generic API that serves a family of plugins is unlikely to be optimized for the use by any one of them.

Relations



Plugins:

- Implement [Monolith](#) or sometimes [Layers](#).
- Extend [Monolith](#) or [Layers](#) with one or two layers of services.

Variants

Plugins are highly variable and omnipresent if we take *Strategy* [GoF] for a kind of plugin:

By the direction of control

The terminology is not settled, but according to what I found over the Web:

- True *Plugins* are registered with and called by the system's *core*, they may call back into the core or return results – they are parts of the system.

- *Addons* are built on top of the system's API and call into the system from outside – they are more like external [Adapters](#) for the system.

By abstractness

A system may use plugins that are more abstract than the core, less abstract or both:

- High-level plugins tend to be related to user experience, statistics, or metadata. They use the core in their own business logic. *Addons* belong here.
- Low-level plugins encapsulate algorithms that are used by the core's business logic.
- Some customizations require multiple plugins: a high-level user-facing addon may rely on algorithms or pieces of business logic implemented by several complementary low-level plugins.

By the direction of communication

A plugin may:

- Provide input to the core (as UI screens and [CLI](#) connections).
- Receive output from the core (e.g. collection of statistics).
- Participate in both input and output (like health check instrumentation).
- Take the role of [Controller](#) (*Orchestrator*) – the plugin processes events from the core and decides on the core's further behavior.
- Be a data processor – the plugin implements a part of a [data processing](#) algorithm which is run by the core. This enables platform-specific optimizations, like SIMD or DSP.

By linkage

Plugins may be *built in* or selected dynamically:

- Every *flavor* (e.g. free, lite, pro, premium) of a product line incorporates a single set of plugins (configuration) built into the application.
- Other systems choose and initiate their plugins on startup according to their configuration files or licenses.
- Still others support attaching and detaching plugins dynamically at runtime.

By granularity

Plugins come in different sizes:

- Small functions or classes are built into the core. They seem to implement the *Strategy* [[GoF](#)] / *Plugins* [[PEAA](#)] design patterns.
- [Aspects](#), such as logging and memory management, pervade a system and are accessed from many places in its code. *Reflection* [[POSA1](#), [POSA4](#)] probably belongs there.
- Modules are plugged in as separate system components. This kind of *Plugins* matches the topic of this book and is further developed by [Hexagonal Architecture](#) and [Microkernel](#).

By the number of instances

A plugin may be:

- *Mandatory* (1 instance), like a piece of algorithm used by the core for a calculation.

- *Optional* (0 or 1 instance), like a smart coloring scheme for a text editor.
- *Subscriptional* (0 or more instances), like log output which may go to a console, the system log, a log file or socket in any combination, or to all of them at once.

By execution mode

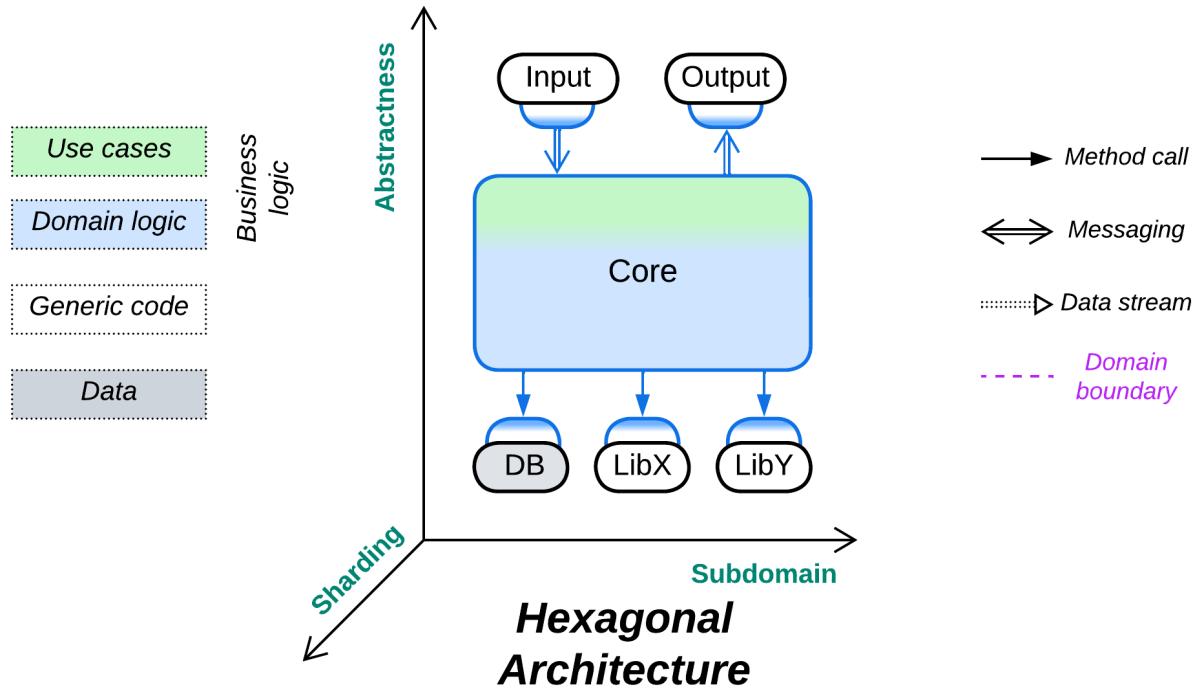
Plugins may be:

- Linked as a binary code called from within the core.
- Written in a *domain-specific language (DSL)* and *interpreted* by the core.
- Communicated with over a network.

Summary

Plugins allow for customization of a component's behavior at the cost of increased complexity, poor testability and somewhat reduced performance.

Hexagonal Architecture



Trust no one. Protect your code from external dependencies.

Known as: Hexagonal Architecture, or originally as Ports and Adapters.

Variants:

By placement of adapters:

- Adapters on the external component's side.
- Adapters on the core side.

Examples – [Hexagonal Architecture](#):

- Hexagonal Architecture / [Ports and Adapters](#),
- DDD-Style Hexagonal Architecture [[DDD](#)] / [Onion Architecture](#) / [Clean Architecture](#).

Examples – [Separated Presentation](#):

- (*Layered*) [Model-View-Presenter](#) (MVP), [Model-View-Adapter](#) (MVA), [Model-View-ViewModel](#) (MVVM), [Model 1](#) (MVC1), Document-View [[POSA1](#)],
- (*Pipelined*) Model-View-Controller (MVC) [[POSA1](#), [POSA4](#)] / [Action-Domain-Responder](#) (ADR) / [Resource-Method-Representation](#) (RMR) / [Model 2](#) (MVC2) / [Game Development Engine](#).

Structure: A monolithic business logic extended with a set of (adapter, component) pairs that encapsulate external dependencies.

Type: Implementation.

Benefits	Drawbacks
Isolates business logic from external dependencies	Suboptimal performance
Facilitates the use of stubs/mocks for testing and development	The vendor-independent interfaces must be designed before the start of development
Allows for qualities to vary between the external components and the business logic	
The programmers of business logic don't	

need to learn any external technologies

References: [Herberto Graça's chronicles](#) is the main collection of patterns from this chapter. *Hexagonal Architecture* has [the original article](#) and a brief summary of its layered variant in [\[LDDD\]](#). Most of the *Separated Presentation* patterns are featured on Wikipedia and there are collections of them from [Martin Fowler](#), [Anthony Ferrara](#) and [Derek Greer](#).

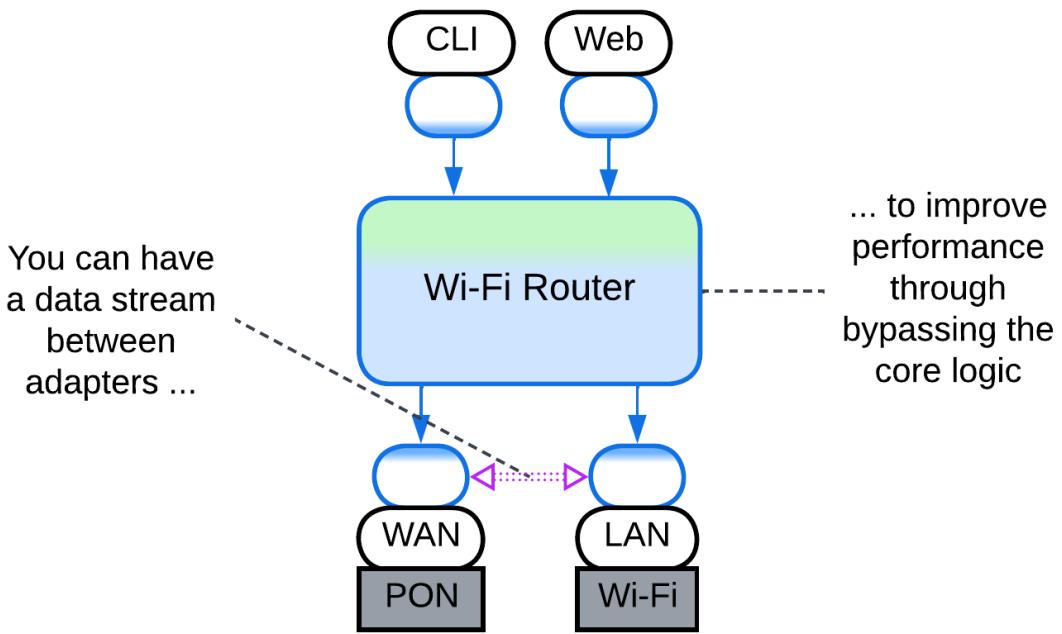
Hexagonal Architecture is a variation of [Plugins](#) that aims for total self-sufficiency of business logic. Any third-party tools, whether libraries, services or databases, are hidden behind [adapters](#) [GoF] that translate the external module's interface into a *service provider interface* (*SPI*) defined by the core module and called *port*. The core's business logic depends only on the ports that its developers defined – a perfect use of [dependency inversion](#) – and manipulates interfaces that were designed in the most convenient way. The benefits of this architecture include the core's cross-platform nature, easy development and testing with [stubs or mocks](#), support for event replay and protection from [vendor lock-in](#). It also allows for replacement of any external library at late stages of the project. The flexibility is paid for with a somewhat longer system design stage and lost optimization opportunities. There is also a high risk to design a leaky abstraction – an SPI that looks generic but whose contract matches that of the component it encapsulates, making it much harder than expected to change the component's vendor.

Stubs and mocks are [test doubles](#) – simplistic replacements for real-world components. They are used to run the business logic in isolation – without the need to deploy any heavyweight libraries or services the logic may depend upon. A *stub* supports a single usage scenario in a single test case while a *mock* is more generic – its behavior is programmed on a per test basis.

Performance

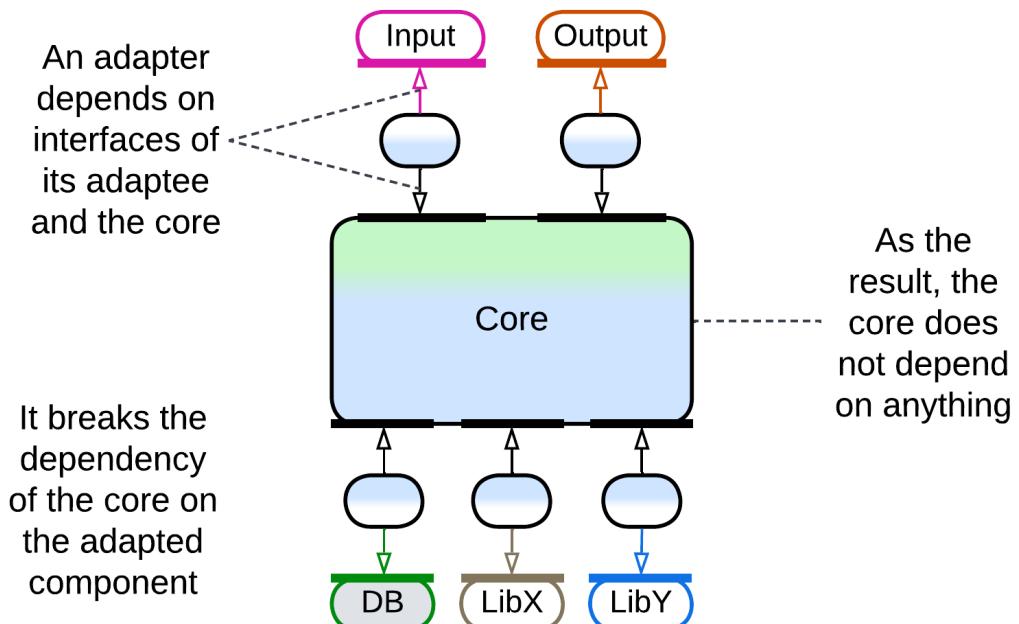
Hexagonal Architecture is a strange beast performance-wise. The generic interfaces (ports) between the core and adapters stand in the way of whole-system optimization and may add context switching. Still, at the same time, each adapter concentrates all the vendor-specific code for its external dependency, which makes the adapter a perfect single place for aggressive optimization by an expert or consultant who is proficient with the adapted third-party software but does not have time to learn the details of your business logic. Thus, some opportunities for optimization are lost while others emerge.

In rare cases the system may benefit from direct communication between the adapters. However, that requires several of them to be compatible or polymorphic, in which case your *Hexagonal Architecture* may in fact be a kind of shallow [Hierarchy](#). Examples include a service that uses several databases which are kept in sync through [Change Data Capture](#) (CDC) or a telephony gateway that interconnects various kinds of voice devices.



Dependencies

Each [adapter](#) breaks the dependency between the core that contains business logic and an adapted component. This makes all the system's components mutually independent – and easily interchangeable and evolvable – except for the adapters themselves, which are small enough to be rewritten as need arises.



Applicability

Hexagonal Architecture benefits:

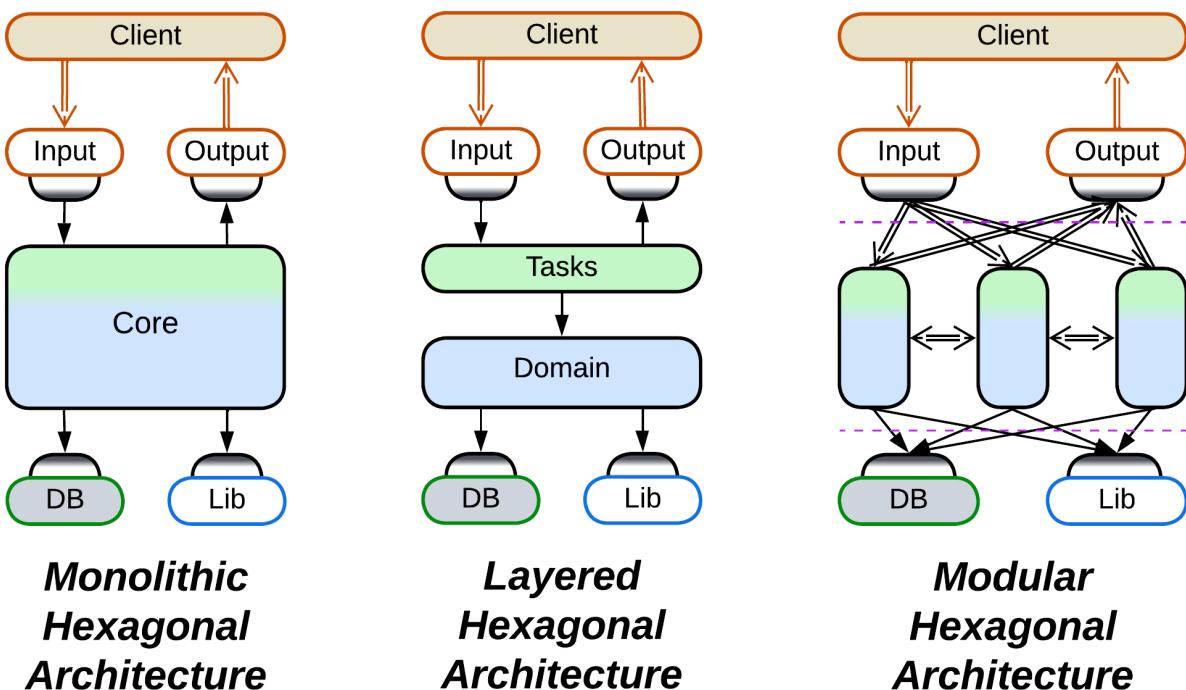
- *Medium-sized or larger components.* The programmers don't need to learn details of external technologies and may concentrate on the business logic instead. The code of the *core* becomes smaller as all the details of managing external components are moved into their *adapters*.

- *Cross-platform development.* The core is naturally cross-platform as it does not depend on any (platform-specific) libraries.
- *Long-lived products.* Technologies come and go, your product remains. Always be ready to change the technologies it uses.
- *Unfamiliar domain.* You don't know how much load you'll need your database to support. You don't know if the library you selected is stable enough for your needs. Be prepared to replace vendors even after the public release of your product.
- *Automated testing.* [Stubs and mocks](#) are great for reducing load on test servers. And stubs for the SPIs which you wrote yourself are easy as a pie.
- *Zero bug tolerance.* SPIs allow for event replay. If your business logic is deterministic, you can [reproduce your user's bugs in your office](#).

Hexagonal Architecture is not good for:

- *Small components.* If there is little business logic, there is not much to protect, while the overhead of defining SPIs and writing adapters is high compared to the total development time.
- *Write-and-forget projects.* You don't want to waste your time on long-term survivability of your code.
- *Quick start.* You need to show the results right now. No time for good architecture.
- *Low latency.* The adapters slow down communication. This is somewhat alleviated by creating direct communication channels between the adapters to bypass the core.

Relations



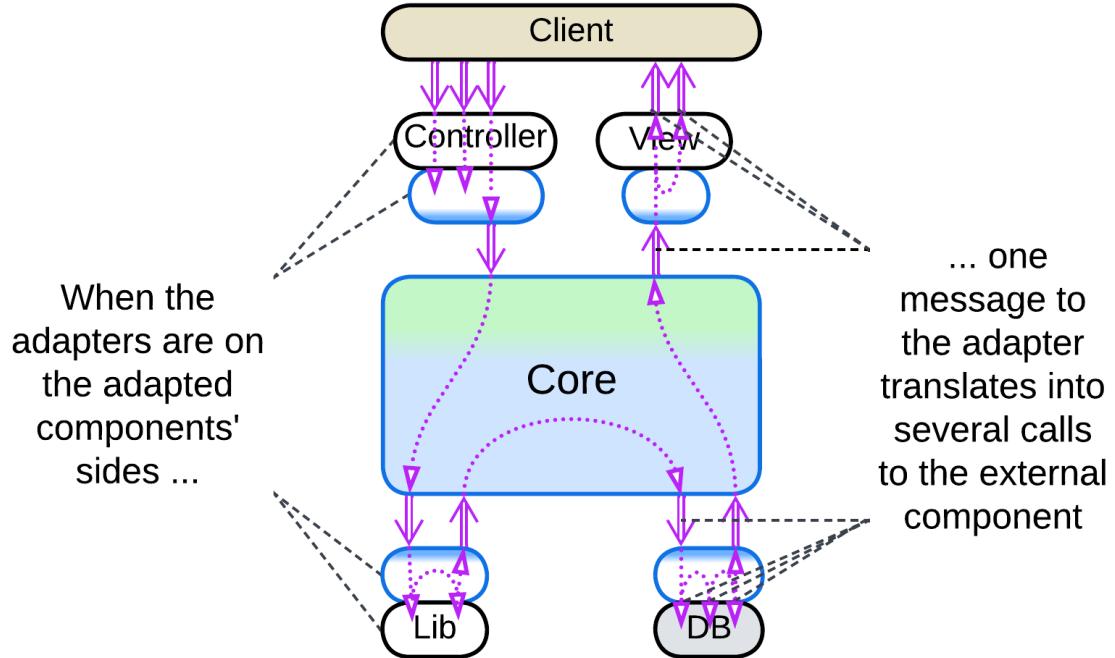
Hexagonal Architecture:

- Is a kind of [Plugins](#).
- May be a shallow [Hierarchy](#).
- Implements [Monolith](#) or [Layers](#).
- Extends [Monolith](#), [Layers](#) or, rarely, [Services](#) with one or two layers of services.
- The [MVC family of patterns](#) is also [derived](#) from [Pipeline](#).

Variants by placement of adapters

One possible variation in a distributed or asynchronous *Hexagonal Architecture* is the deployment of [adapters](#), which may reside adjacent to the core or with the components they adapt:

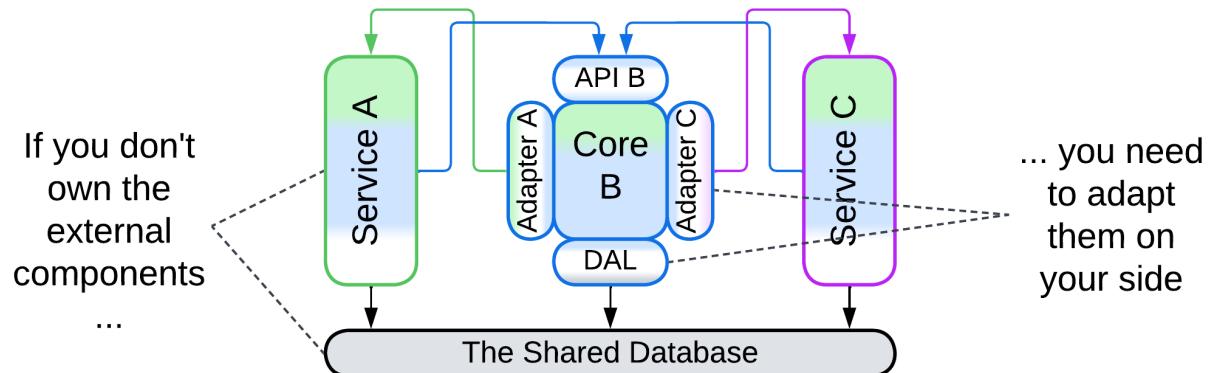
Adapters on the external component side



If your team owns the component adapted, the *adapter* may be placed next to it. That usually makes sense because a single domain message (in the terms of your business logic) tends to unroll into a series of calls to an external component. The fewer messages you send, the faster your system is.

This resembles [Sidecar \[DDS\]](#) and [Open Host Service \[DDD\]](#).

Adapters on the core side



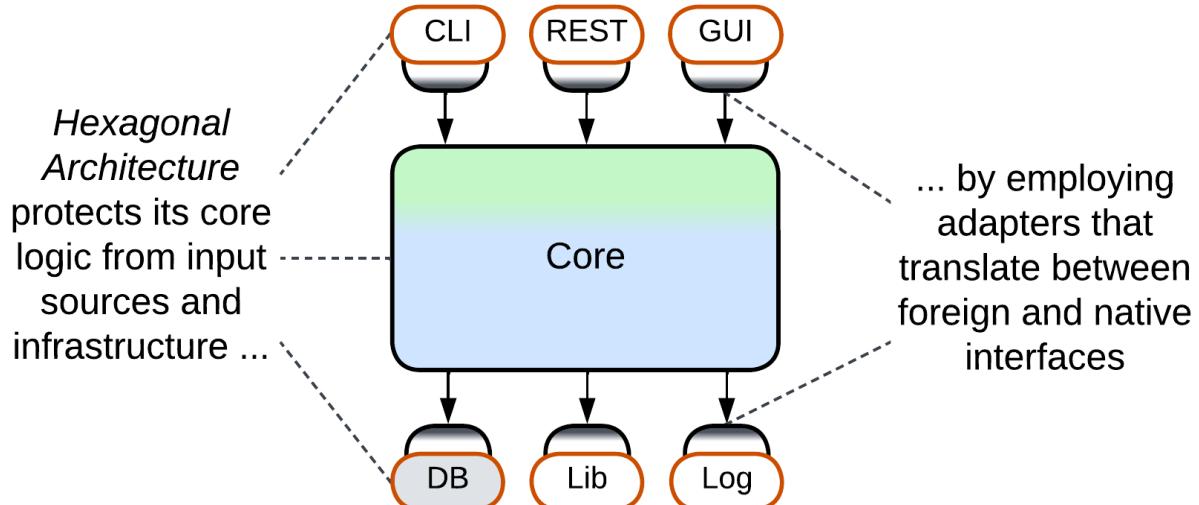
Sometimes you need to adapt an external service which you don't control. In that case the only real option is to place its *adapter* together with your *core* logic. In theory, the adapter can be deployed as a separate component, maybe in a [Sidecar \[DDS\]](#), but that may slow down communication.

This approach resembles [Ambassador \[DDS\]](#) and [Anticorruption Layer \[DDD\]](#).

Examples – Hexagonal Architecture

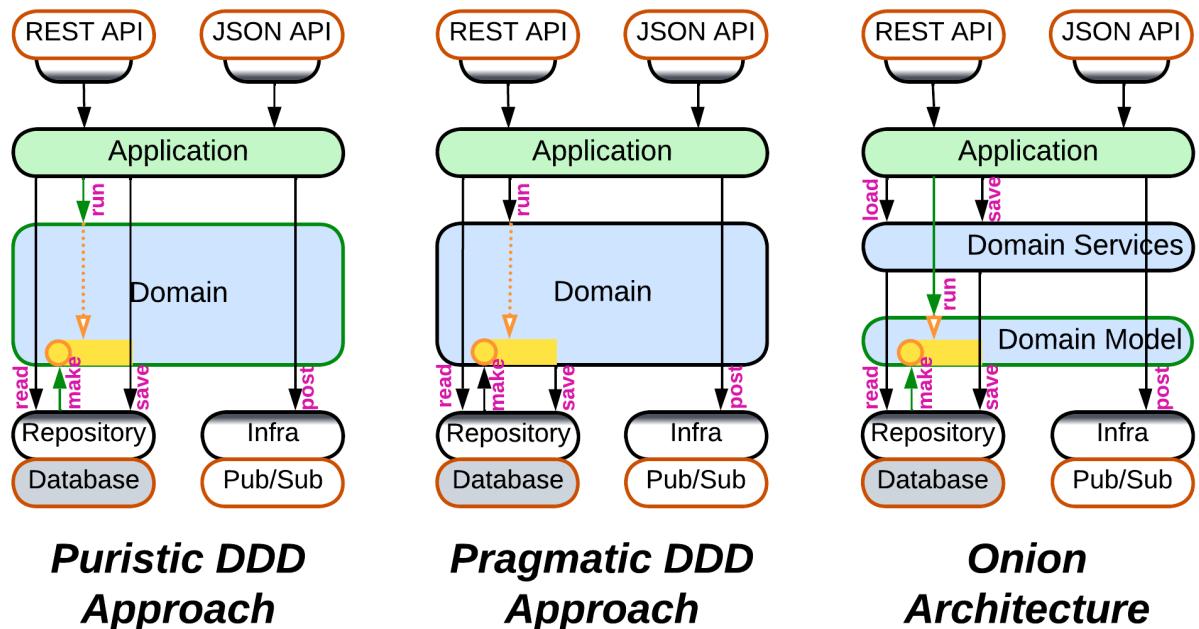
Hexagonal Architecture protects business logic from all its dependencies. It is simple and unambiguous. It does not come in many shapes:

Hexagonal Architecture, Ports and Adapters



Just like [MVC](#) it is based on, the original *Hexagonal Architecture (Ports and Adapters)* does not care about the contents or structure of its core – it is all about isolating the core from the environment. The core may have layers or modules or even plugins inside, but the pattern has nothing to say about them.

DDD-Style Hexagonal Architecture, Onion Architecture, Clean Architecture



As *Hexagonal Architecture* built upon the [DDD](#)'s idea of isolating business logic with [Adapters](#), it was quickly integrated back into DDD [[LDDD](#)]. However, as [Ports and Adapters](#)

appeared later than the [original DDD book](#), there is no universal agreement on how the thing should work:

- The cleanest way is for the *domain* layer to have nothing to do with the database – with this approach the *application* asks the [*repository*](#) (the database *adapter*) to create *aggregates* (domain objects), then executes its business actions on the aggregates and tells the repository to save the changed aggregates back to the database.
- Others say that in practice the logic inside an aggregate may have to read additional information from the database or even depend on the result of persisting parts of the aggregate. Thus it is the aggregate, not the application, which should save its changes, and the logic of accessing the database leaks into the domain layer.
- [*Onion Architecture*](#), one of early developments of *Hexagonal Architecture* and DDD, always splits the domain layer into a *domain model* and a *domain services*. The *domain model* layer contains classes with business data and business logic, which are loaded and saved by the *domain services* layer just above it. And the upper *application services* layer drives use cases by calling into both domain services and domain model.
- There is also [*Clean Architecture*](#) which seems to generalize the approaches above without delving into practical details – thus the way it saves its aggregates remains a mystery.

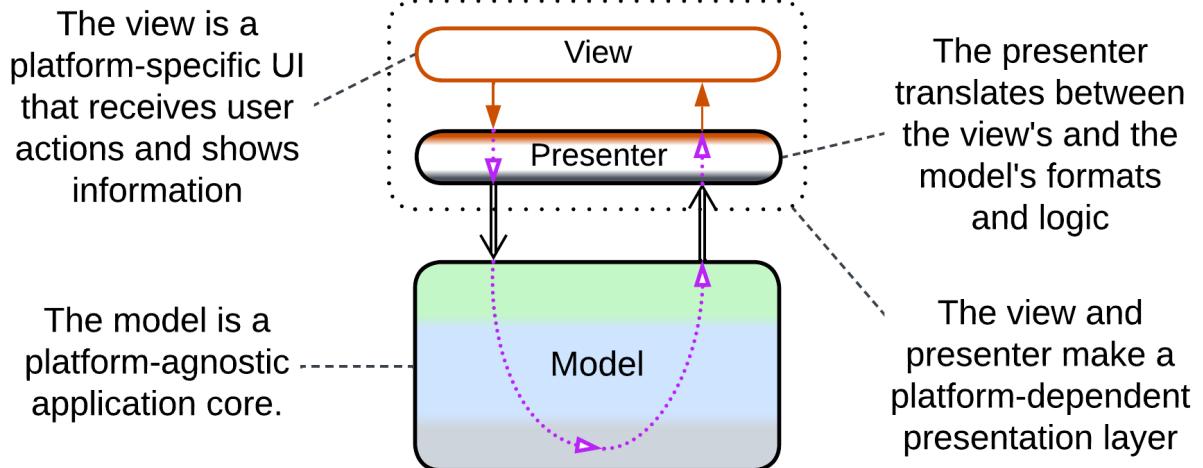
Examples – Separated Presentation

[*Separated Presentation*](#) protects business logic from a dependency on *presentation* (interactions with the system's user via a window, command line, or web page). There is a [*great variety*](#) of such patterns, commonly known as *Model-View-Controller* (MVC) alternatives. They are derived from *Hexagonal Architecture* by omitting every component not directly involved in user interactions and make three structurally distinct groups:

- Bidirectional flow – the *view* (user-facing component) both receives input and produces output and there is often an explicit *adapter* between it and the main system, resulting in [*Layers*](#).
- Unidirectional flow – the *controller* receives input while the *view* produces output, [*forming*](#) a kind of [*Pipeline*](#).
- Hierarchical with multiple *models*, [*discussed*](#) in the [*Hierarchy*](#) chapter.

All of them aim at making the business logic presentation-agnostic (thus cross-platform and developed by an independent team), but differ in their complexity, flexibility and best use cases.

Model-View-Presenter (MVP), Model-View-Adapter (MVA), Model-View-ViewModel (MVVM), Model 1 (MVC1), Document-View



MVP-style patterns pass user input and output through one or more presentation [layers](#). Each pattern includes:

- *View* – the interface exposed to users.
- An optional intermediate layer that translates between the *view* and *model*. It is the component which differentiates the patterns, both in name and function.
- *Model* – the whole system's business logic and infrastructure, now independent from the method of presentation (CLI, UI or web).

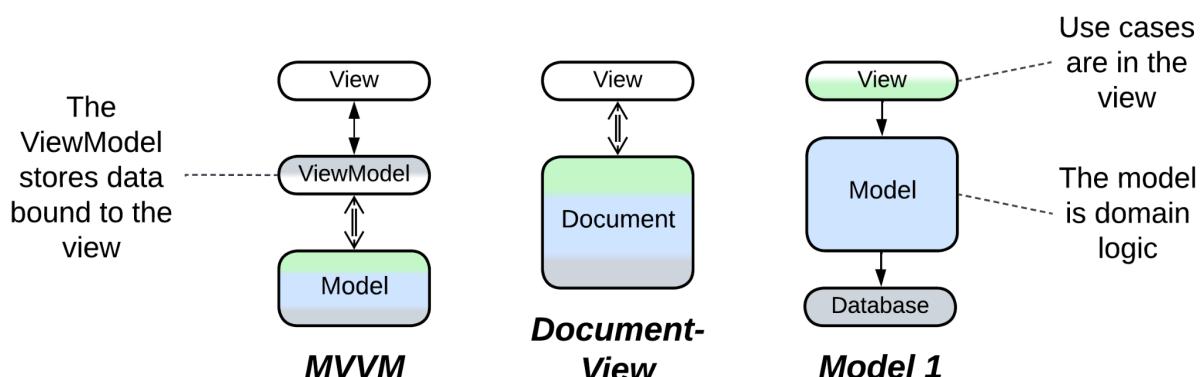
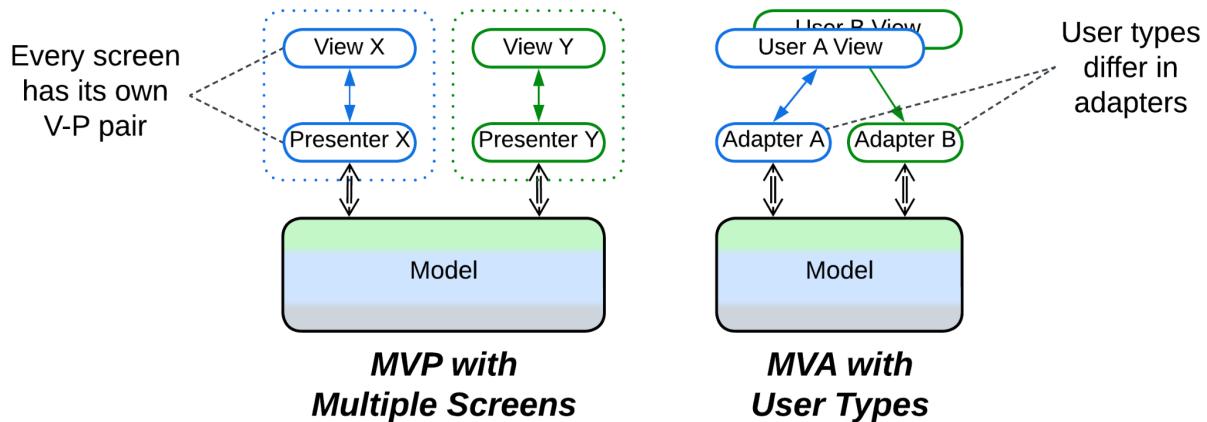
Document-View [[POSA1](#)] and *Model 1* (MVC1) skip the intermediate layer and connect the view directly to the *model* (*document*). These are the simplest [Separated Presentation](#) patterns for UI and web applications, correspondingly.

In a [Model-View-Presenter](#) (MVP), the *presenter* ([Supervising Controller](#)) receives input from the *view*, interprets it as a call to one of the *model*'s methods, retrieves the call's results and shows them in the *view*, which is often completely dumb ([Passive View](#)). A complex system may feature multiple view-presenter pairs, one per UI screen.

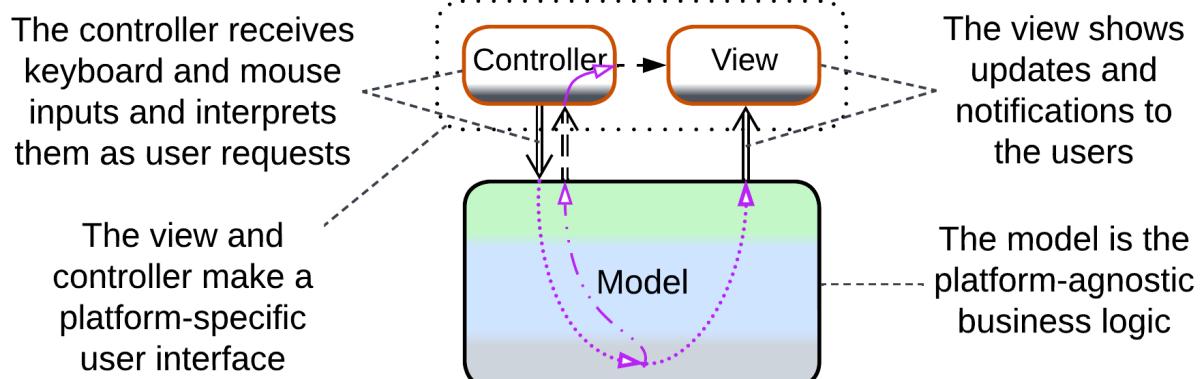
A [Model-View-Adapter](#) (MVA) is quite similar to MVP, but it chooses the adapter on a per session basis while reusing the view. For example, an unauthorized user, a normal user, and an admin would access the *model* through different adapters that would show them only the data and actions available with their permissions.

A [Model-View-ViewModel](#) (MVVM) uses a stateful intermediary (*ViewModel* or *Presentation Model*) which resembles a [Response Cache](#), [Materialized View](#), [Reporting Database](#) or *Read Model* of [CQRS](#) – it stores all the data shown in the *view* in a form which is convenient for the *view* to [bind to](#). Changes in the *view* are propagated to the *ViewModel* which translates them into requests to the underlying application (the true *model*). Changes in the *model* (independent or resulting from user actions) are propagated to the *ViewModel* and, eventually, to the *view*.

All those patterns exploit modern OS or GUI frameworks' widgets which handle and process mouse and keyboard input, thus [removing](#) the need for a separate (input) *controller* (see below).



Model-View-Controller (MVC), Action-Domain-Responder (ADR), Resource-Method-Representation (RMR), Model 2 (MVC2), Game Development Engine



When your presentation's input and output diverge (raw mouse movement vs 3D graphics in UI, HTTP requests vs HTML pages in websites), it makes sense to separate the presentation layer into dedicated components for input and output.

Model-View-Controller (MVC) [[POSA1](#), [POSA4](#)] allows for cross-platform development of hand-crafted UI applications (which was necessary before universal UI frameworks emerged) by abstracting the system's *model* (its main logic and data, the core of *Hexagonal Architecture*) from its user interface containing platform-specific *controller* (input) and *view* (output):

- The *controller* translates raw input into calls to the business-centric model's API. It may also hide or lock widgets in the view when the model's state changes.

- The *model* is the main UI-agnostic application which executes controller's requests and notifies the view and, optionally, controller when its data changes.
- Upon receiving a notification, the *view* reads, transforms, and presents to the user the subset of the model's data which it covers.

Each widget on the screen may have its own model-view pair. The absence of an intermediate layer between the view and model makes the view heavyweight as it has to translate the model's data format into something presentable to users – the flaw addressed by the *MVP* (3-layered) patterns discussed above.

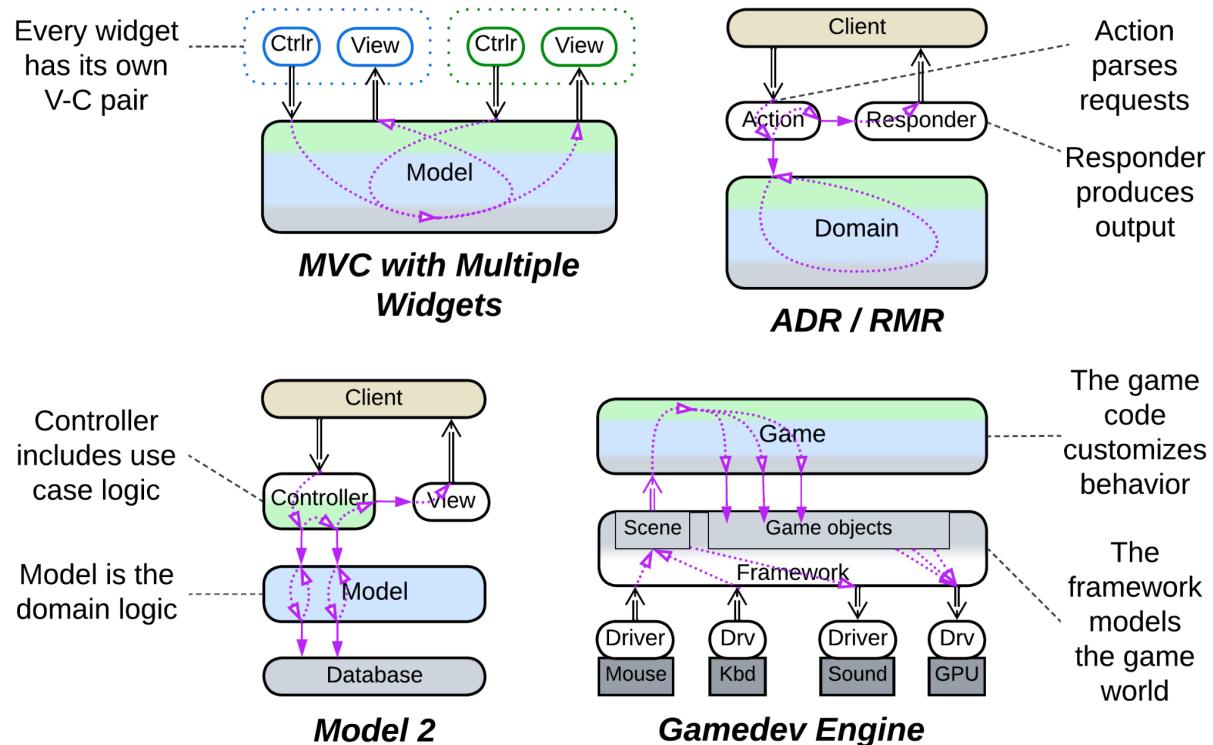
Both *Action-Domain-Responder* (*ADR*) and *Resource-Method-Representation* (*RMR*) are web layer patterns. An *action* (*method*) receives a request, calls into a *domain* (*resource*) to make changes and retrieve data and brings the results to a *responder* (*representation*) which prepares the return message or web page. *ADR* is technology-agnostic while *RMR* is HTTP-centric.

Model 2 (*MVC2*) is a similar pattern from the Java world with integration logic implemented in the controller.

A *game development engine* creates a higher-level abstraction over input from mouse / keyboard / joystick and output to sound card / GPU while more powerful engines may also model physics and character interactions. The role is quite similar to what the original *MVC* did, with a couple of differences:

- Games often have to deal with the low-level and very chatty interfaces of hardware components, thus the input and output are at the bottom side of the system diagram.
- The framework itself makes a cohesive layer, becoming a kind of *Microkernel*.

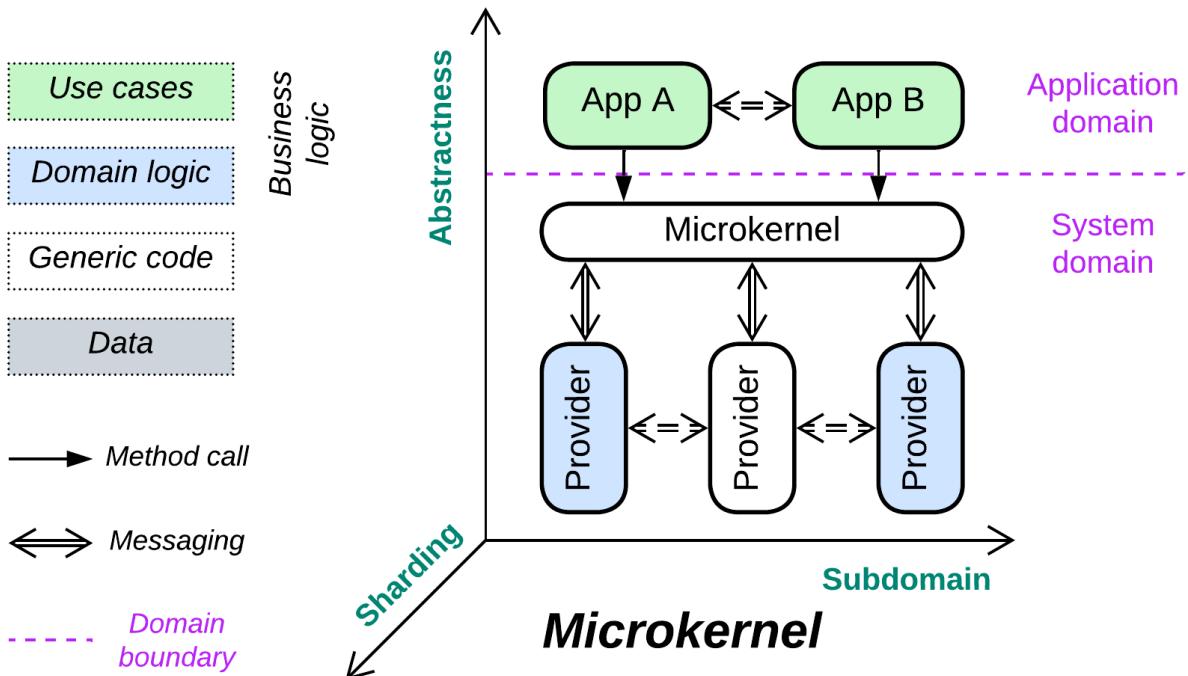
Another difference is that while *MVC* provides for changing target platforms by rewriting its minor components (*view* and *controller*), you are very unlikely to change your game framework – instead, it is the framework itself that makes all the platforms look identical to your code.



Summary

Hexagonal Architecture isolates a component's business logic from its external dependencies by inserting *adapters* between them. It protects from *vendor lock-in* and allows for late changes of third-party components but requires all the APIs to be designed before programming can start and often hinders performance optimizations.

Microkernel



Communism. Share resources among consumers.

Known as: Microkernel [[POSA1](#), [POSA4](#) but [not SAP](#) and [FSA](#)].

Aspects:

- [Middleware](#),
- [Orchestrator](#).

Variants:

- Operating System,
- Software Framework,
- Virtualizer / Hypervisor / Container Orchestrator [[DDS](#)] / Distributed Runtime,
- Interpreter [[GoF](#)] / Script / Domain-Specific Language (DSL),
- Configurator / Configuration File,
- Saga Engine,
- [AUTOSAR Classic Platform](#).

Structure: A layer of [Orchestrators](#) over a [Middleware](#) over a layer of [Services](#).

Type: Implementation.

Benefits	Drawbacks
The system's complexity is evenly distributed among the components	The API and SPIs are very hard to change
Polymorphism of the resource providers	Performance is suboptimal
The components can have independent qualities	Latency is often unpredictable
A resource provider can be implemented and tested in isolation	
Each application is sandboxed by the microkernel	
The system is platform-independent	

References: Microkernel pattern in [POSA1].

While vanilla [Plugins](#) and [Hexagonal Architecture](#) keep the business logic in the monolithic *core* component, *Microkernel* treats the core as a thin [Middleware](#) (called *microkernel*) that connects user-facing applications (*external services*) to resource providers (*internal services*). The *resource* in question can be anything ranging from CPU time or RAM to business functions. The external services communicate with the microkernel through its *API* while the internal services implement the microkernel's *service provider interfaces* (*SPIs*) (usually there is a kind of internal service and an SPI per resource type).

On one hand, the pattern is very specific and feels esoteric. On the other – it is indispensable in many domains, with many more real-life occurrences than would be expected. *Microkernel* finds its place where there are a variety of applications that need to use multiple shared resources, with each resource being independent of others and requiring complex management.

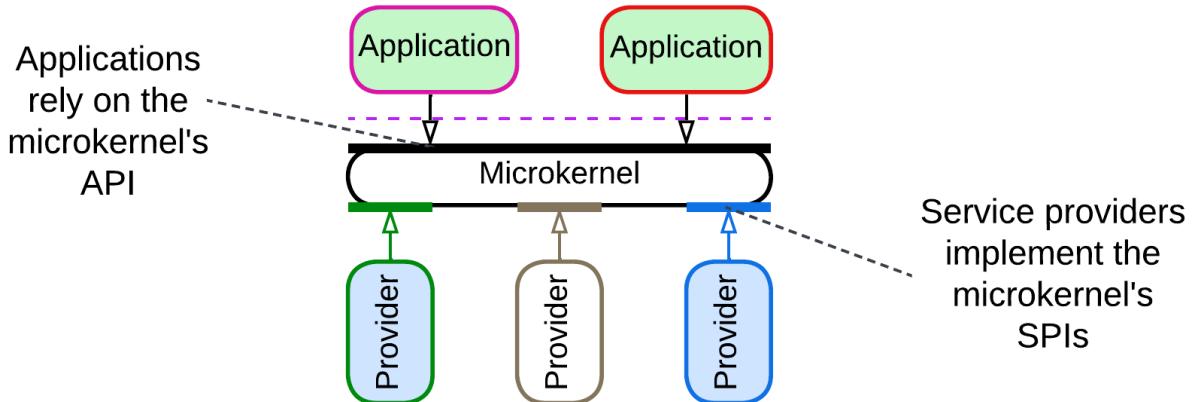
Performance

The *microkernel*, being an extra layer of indirection, degrades performance. The actual extent varies from a few percent for OSes and *virtualizers* to an order of magnitude for *scripts*. A more grievous aspect of performance is that latency becomes unpredictable as soon as the system runs short of one of the shared resources: memory, disk space, CPU time, or even storage for deleted objects. That is why [real-time systems](#) rely on minimalistic [real-time OSes](#) or even run on bare metal.

It is common to see system components communicate directly via shared memory or sockets bypassing the *microkernel* to alleviate the performance penalty it introduces.

Dependencies

The *applications* depend on the *API* of the *microkernel* while the *providers* depend on its *SPIs*. On one hand, that isolates the applications and providers from each other, letting them develop independently. On the other hand, the microkernel's API and SPIs should be very stable to support older versions of the components which the microkernel integrates.



Applicability

Microkernel is [applicable](#) in:

- *System programming*. You manage system resources and services which will be used by untrusted client applications. Hide the real resources behind a trusted proxy

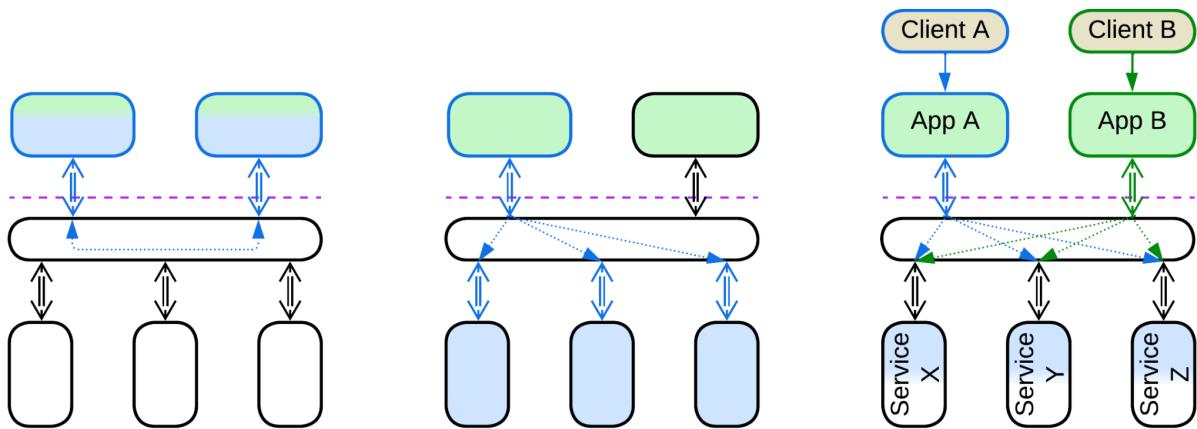
layer. Be ready to change the hardware platform without affecting existing client code.

- *Frameworks that integrate several subdomains.* The microkernel component coordinates multiple specialized libraries. Its API is a *Facade* [GoF] for the managed functionality.
- *Scripting or DSLs.* The microkernel is an *Interpreter* [GoF] which lets your clients' code manage the underlying system.

Microkernel does not fit:

- *Coupled domains.* Any degree of coupling between the resource providers complicates the microkernel and its SPIs and is likely to degrade performance which, however, may be salvaged by introducing direct communication channels between the providers.

Relations



Microkernel as a Middleware

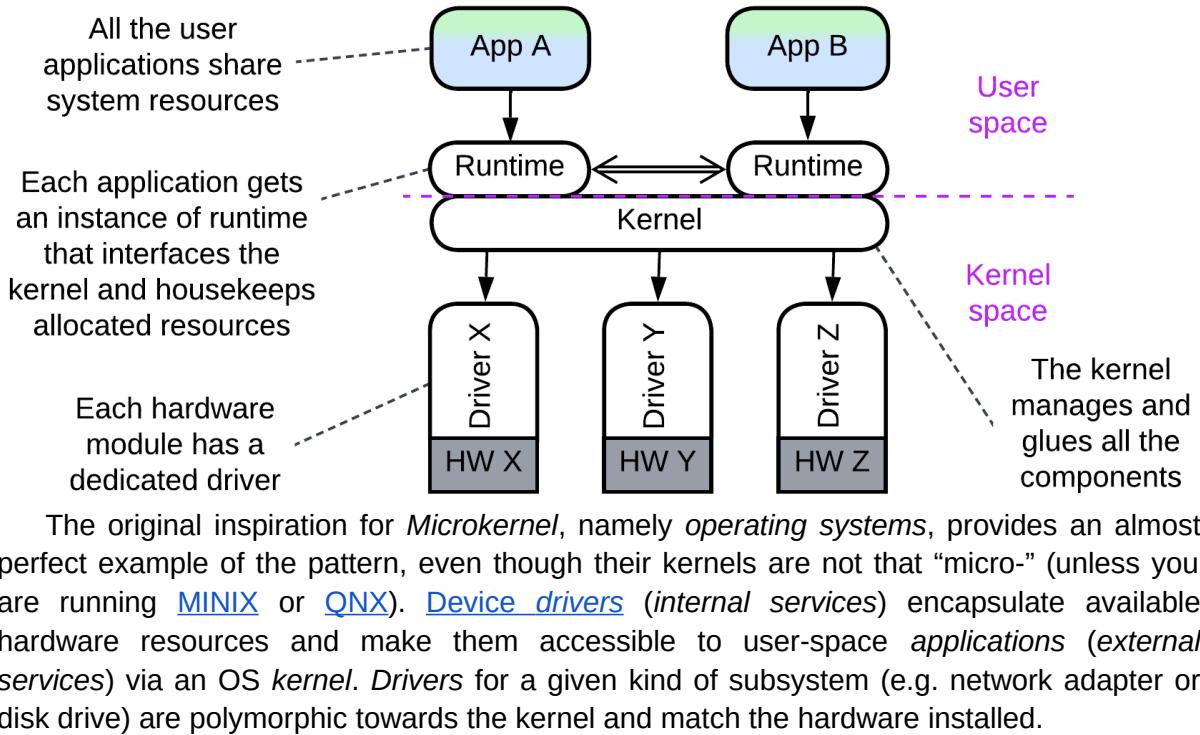
Microkernel:

- Is a specialization of [Plugins](#).
- Is related to [Backends for Frontends](#), which is a layer of [Orchestrators](#) over a layer of [Services](#); *Microkernel* adds a [Middleware](#) in between.
- Is a kind of 2-layered [SOA](#) with an [ESB](#).
- The *microkernel* layer serves as (implements) a [Middleware](#) for the upper (*external*) [Services](#) and often makes an [Orchestrator](#) for the lower (*internal*) [Services](#).
- May be implemented by [Mesh](#).

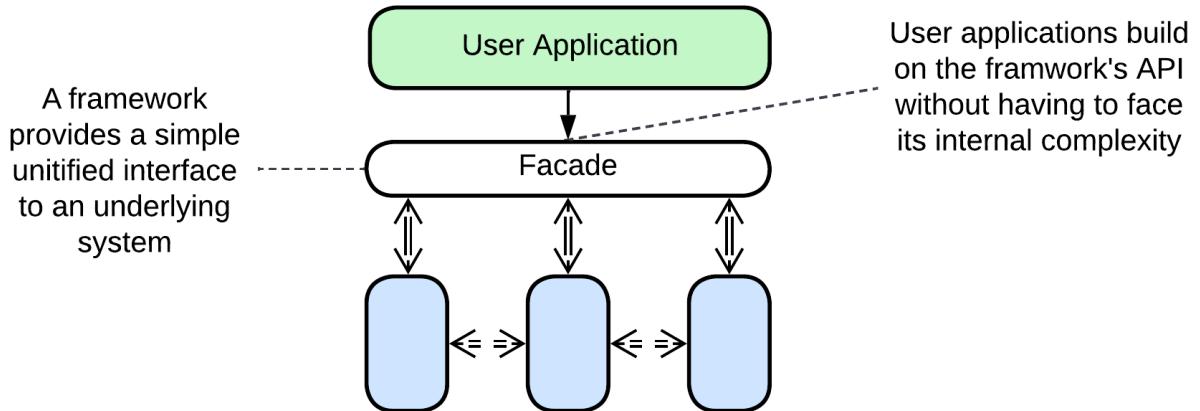
Variants

Microkernel can appear in many forms:

Operating System

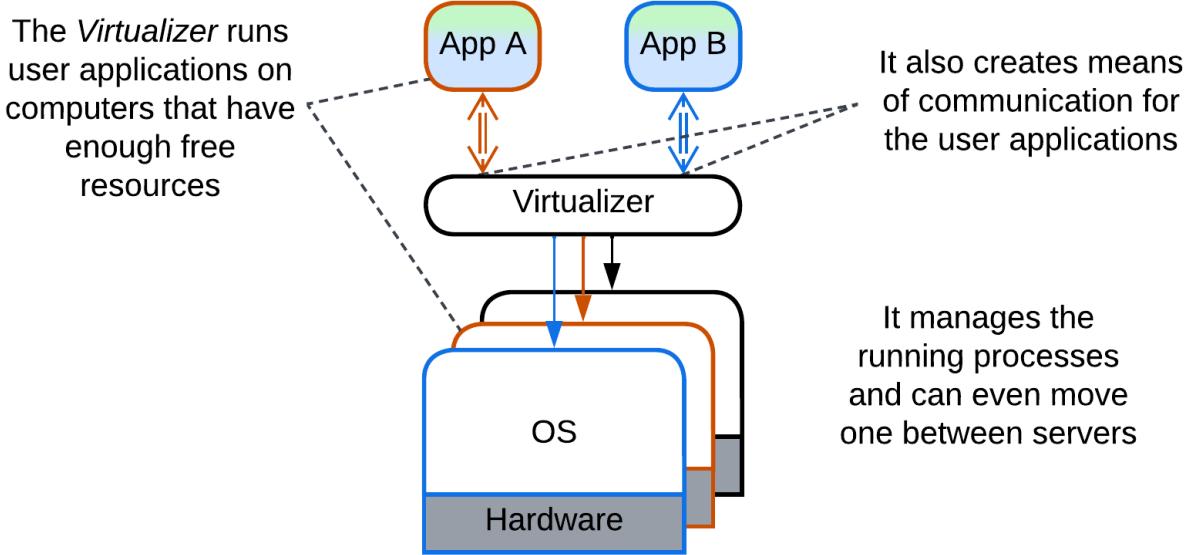


Software Framework



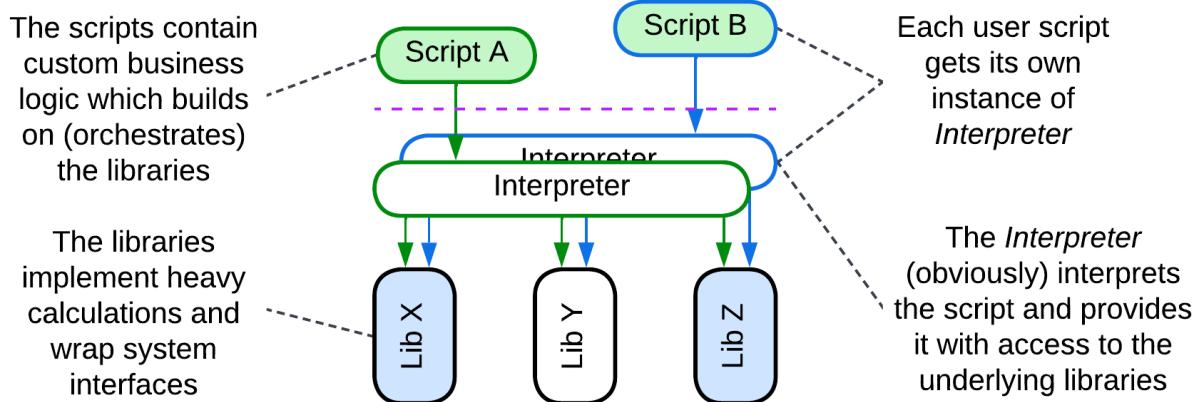
The *microkernel* is a [Facade](#) [GoF] that integrates a set of libraries and exposes a user-friendly high-level interface. [PAM](#) looks like a reasonably good example.

Virtualizer, Hypervisor, Container Orchestrator, Distributed Runtime



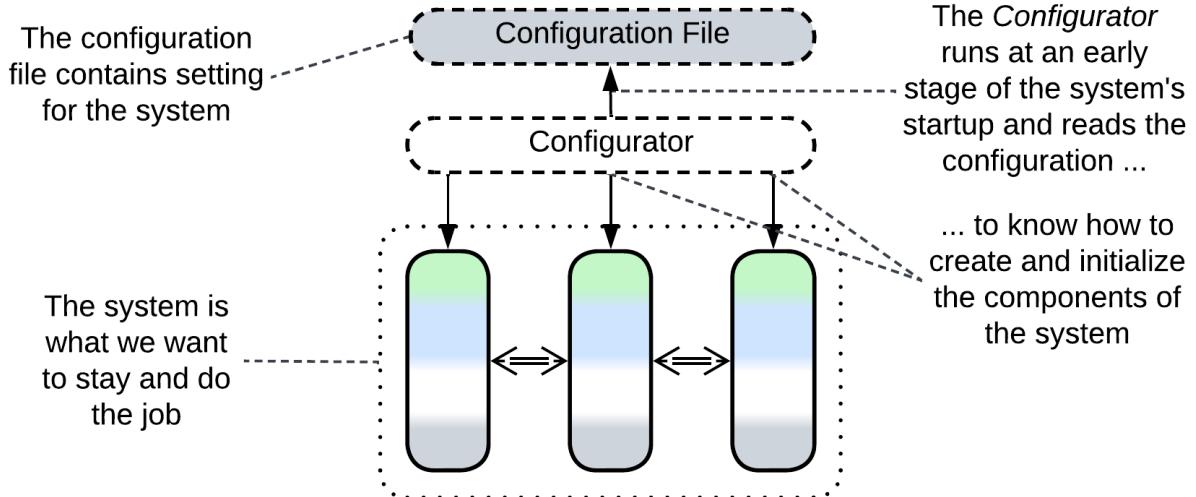
Hypervisors (Xen), PaaS and [FaaS](#), container orchestrators (Kubernetes) [[DDS](#)], and distributed actor frameworks (Akka, Erlang/Elixir/OTP) use resources of the underlying computer(s) to run guest applications. A hypervisor virtualizes the resources of a single computer while a distributed runtime manages those of multiple servers – in the last case there are several instances of the same kind of an *internal server* which abstracts a host system.

Interpreter, Script, Domain-Specific Language (DSL)



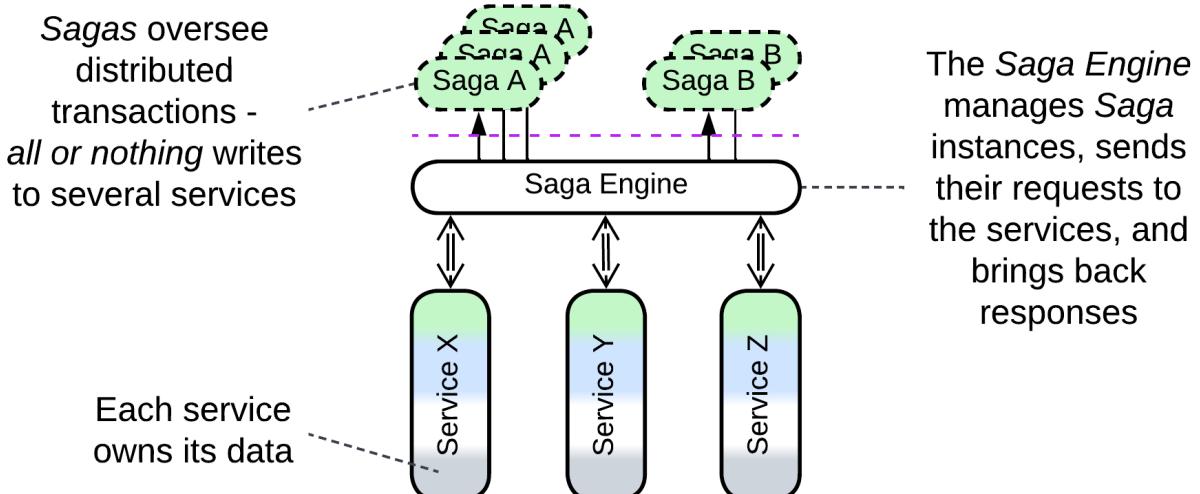
User-provided scripts are run by an *Interpreter* [[GoF](#)] which also allows them to access a set of installed libraries. The *Interpreter* is a microkernel, and the syntax of the script or [DSL](#) it interprets is the microkernel's API.

Configurator, Configuration File



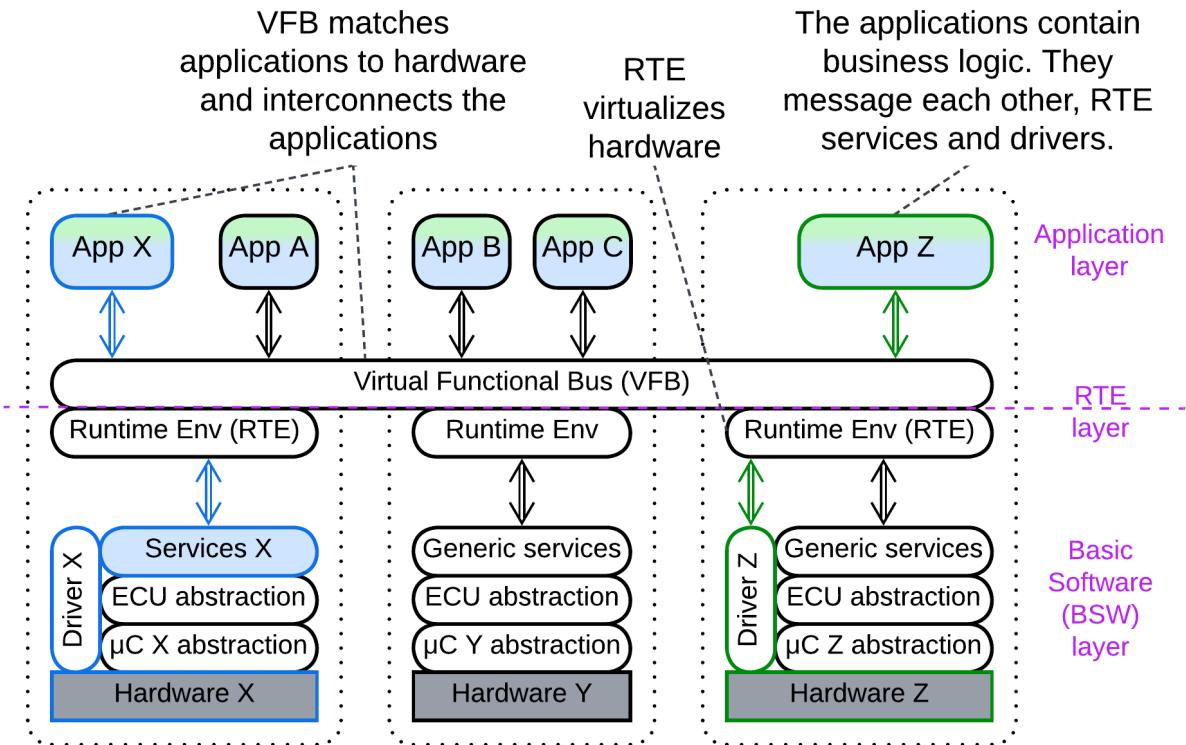
Configuration files may be regarded as short-lived scripts that configure the underlying modules at the start of the system. The parser of the configuration file is a transient *microkernel*.

Saga Engine



A [Saga \[MP\]](#) orchestrates distributed transactions. It may be written in a *DSL* which requires a compiler or interpreter, which is a *microkernel*, to execute.

AUTOSAR Classic Platform



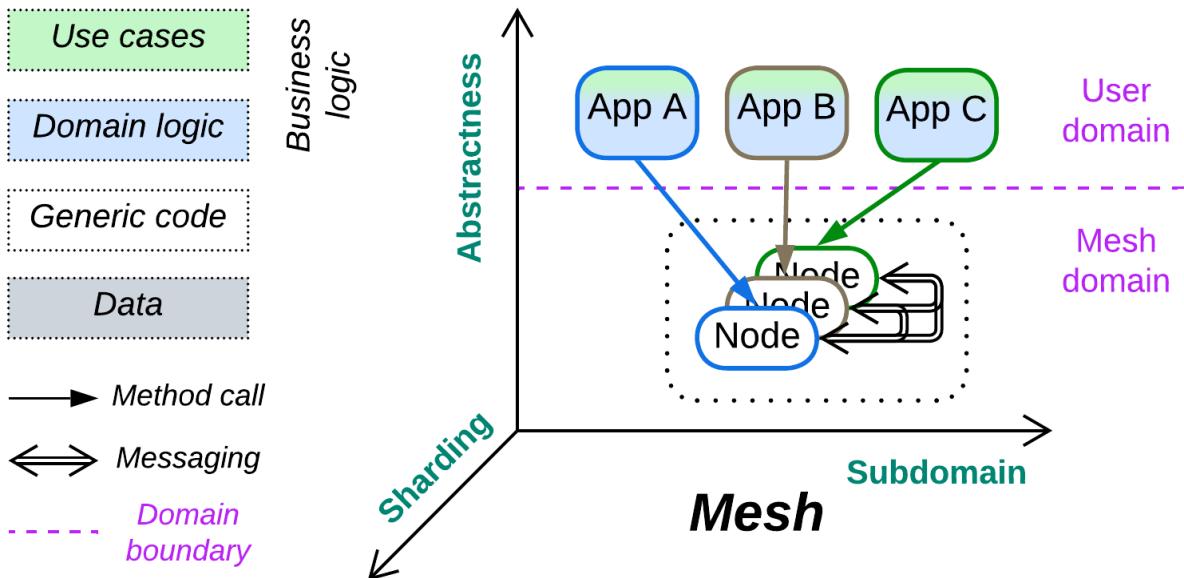
The [notorious automotive standard](#), though promoted as [SOA](#), is structured as a distributed / virtualized *Microkernel*. The application layer comprises a network of *software components* spread out over hundreds of chips which are, for some secret reason, called *electronic control units (ECUs)*. The communication paths between the software components and much of the code are static (auto-generated at compilation time). A software component may access hardware of its ECU via standard interfaces.

The *microkernel* shows up as *Virtual Functional Bus (VFB)* which, as a *distributed Middleware*, provides communication between the *applications* by virtualizing multiple *Runtime Environments (RTEs)* – the local [system interfaces](#).

Summary

Microkernel is a ubiquitous approach to sharing resources among consumers, where both resource providers and consumers may be written by external companies.

Mesh



Hive mind. Go decentralized.

Known as: Mesh, Grid.

Aspects: those of [Middleware](#).

Variants: Meshes vary greatly. Examples include:

- [Peer-to-Peer Networks](#),
- Leaf-Spine Architecture / [Spine-Leaf Architecture](#),
- Actors,
- [Service Mesh](#) [[FSA](#), [MP](#)],
- [Space-Based Architecture](#) [[SAP](#), [FSA](#)].

Structure: A system of interconnected [Shards](#) which usually make a [Middleware](#).

Type: Implementation.

Benefits	Drawbacks
No single point of failure	Overhead in administration and security
The system is able to self-heal	Performance is likely to suffer
Great scalability	The <i>Mesh</i> itself is very hard to debug
Available off the shelf	Unreliable communication must be accounted for in the code

References: [Wikipedia](#) and [DDIA](#) on topology and protocols. [\[FSA\]](#) on Service Mesh and Space-Based Architecture. A [long](#) and [short](#) article on Service Mesh.

If a system is required to survive faults, all of its components must be both [sharded](#) and interconnected, which makes a *Mesh* – a network of interacting instances (*nodes*). In most cases the lower layer of a *shard* implements connectivity while the business logic resides in its upper layer(s). Whilst the connectivity component tends to be identical in every node of a system, the upper components may be identical – forming [Shards](#), or different – forming [Services](#).

Most *Meshe*s support adding and removing parts of their networks dynamically, which allows for scaling up, scaling down, and fault recovery. That is achieved through a flexible network topology, which has the chance of missing or duplicating requests, which may lead

to a single action being executed by two instances of a service in parallel or by the same instance twice. Moreover, *Mesh*-mediated communication is likely to be slower than direct one.

Performance

In most (all?) implementations the user *application* is colocated with a *node* of the *Mesh*, thus communicating through the *Mesh* does not add an extra network hop (which would strongly degrade performance). However, that holds true only when the *Mesh node* knows the destination of the message it should send – when it has already established a communication channel towards it. Finding a new destination may not always be easy and would often require consulting registries and sometimes waiting for the network topology to stabilize, which may involve timeouts (like the ones you could have experienced with torrents). On the other hand, no other architecture is known to seamlessly support huge networks.

Dependencies

Mesh, being a *sharded Middleware*, inherits dependencies from both of its parent metapatterns:

- As with [Middleware](#), the services that run over a *Mesh* depend both on the *Mesh*'s API and on each other (or on a shared message format, aka [Stamp Coupling](#), or a [Shared Database](#) if they [use one for communication](#)).
- As with [Shards](#), the nodes of the *Mesh* should communicate through a backward- and forward-compatible protocol as there will likely be periods of time when multiple versions of the *Mesh* nodes coexist.

Applicability

Mesh is perfect for:

- *Dynamic scaling*. Instances of services may be quickly added or removed.
- *High availability*. A *Mesh* is very hard to disable or kill because it both creates new instances of failed services and finds routes around failed connections.

Mesh fails in:

- *Low latency domains*. Spreading information through a *Mesh* is slow and sometimes unreliable.
- *Security-critical systems*. A public *Mesh* exposes a high attack surface while the scalability of private deployments is limited by the installed hardware.
- *Quick and dirty programming*. The possible message duplication may cause evil bugs if you ignore the risks.

Relations

Mesh:

- Misuses [Shards](#).
- Uses [Layers](#).
- Is the base for running multiple instances of a [Monolith](#), [Layers](#), or [Services](#).
- Implements a distributed [Middleware](#), [Shared Repository](#), or [Microkernel](#).

Variants

Meshes are known to vary:

By structure

The connections in a *Mesh* may be:

- *Structured* or *pre-defined* – the *Mesh* is pre-designed and hard-wired. This kind of topology provides redundancy but not scalability.
- *Unstructured* or *ad-hoc* – nodes can be added and removed at runtime, causing restructuring of the *Mesh*.

By connectivity

Each *node* is:

- Connected to all other nodes – a *fully connected Mesh*. Such *Meshes* are limited in size because the number of interconnections grows as a square of the number of nodes. Notwithstanding, they offer the best communication speed and delivery guarantees.
- Connected to some other nodes. There are many possible *topologies* with the correct choice for any given task better left to experts.

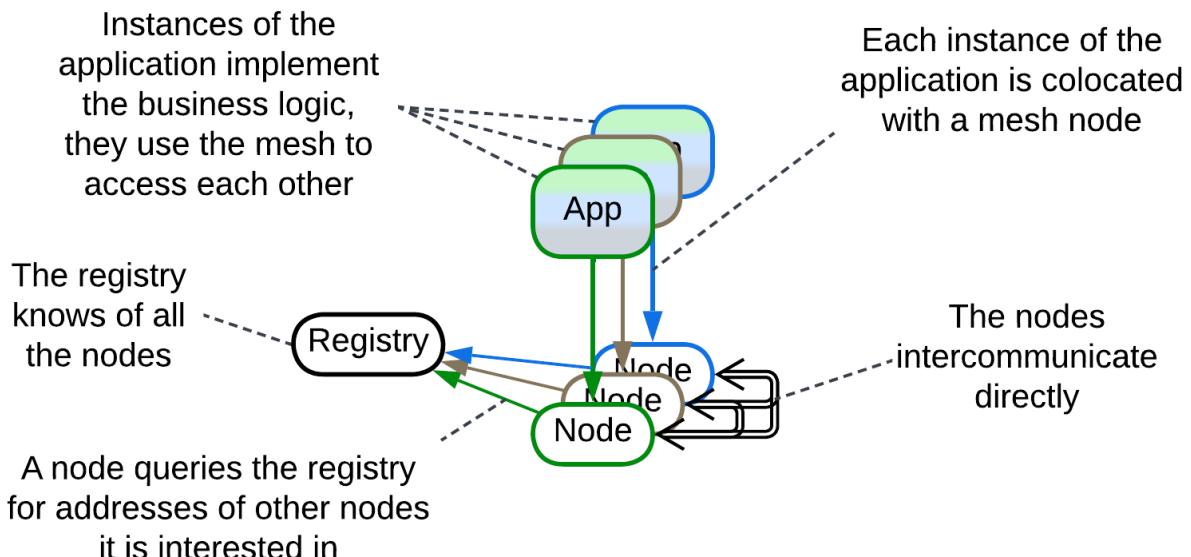
By the number of mesh layers

The connected *nodes* of a *Mesh* may be:

- Identical (one-layer *Mesh*). A node behaves according to its site in the network.
- Specialized (multi-layer *Mesh*). Some nodes implement *trunk* (route messages and control the topology) while others are *leaves* (run user applications).

Examples

Peer-to-Peer Networks

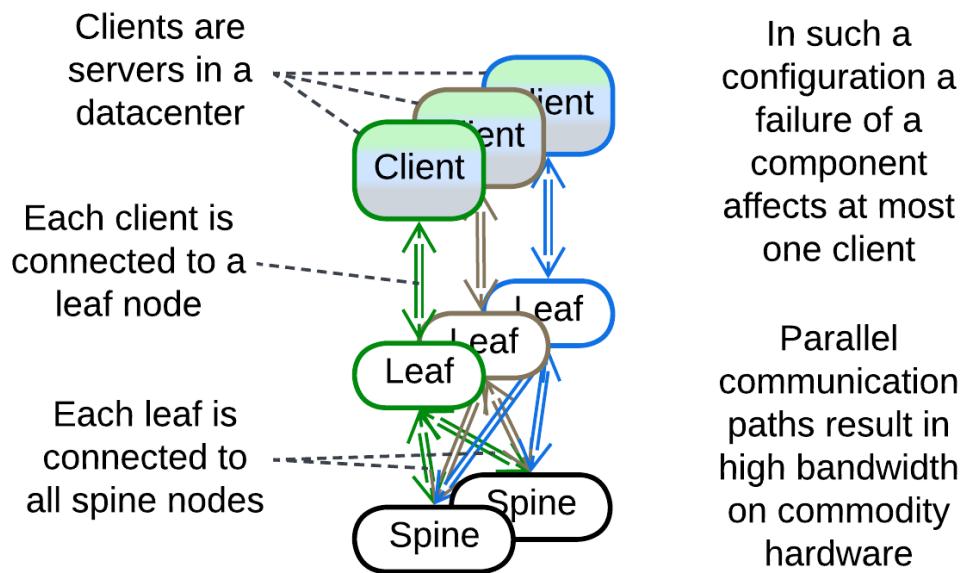


[Peer to Peer](#) (P2P) networks are intended for massive resource sharing over unstable connections. The resource in question may be data (torrents, blockchain, [P2PTV](#)), CPU time

([volunteer computing](#), [distributed compilation](#)), or Internet access ([Tor](#), [I2P](#)). In most cases it is shared over an *unstructured* (as participants join and leave) 2-layer (there are dedicated servers that register and coordinate users) network which is *overlaid* on top of the Internet. All the leaf nodes run identical narrowly specialized software (i.e. either file sharing or blockchain but not both at once) which provides the clients with access to resources of other nodes, making a kind of distributed [Middleware](#) or [Shared Repository](#).

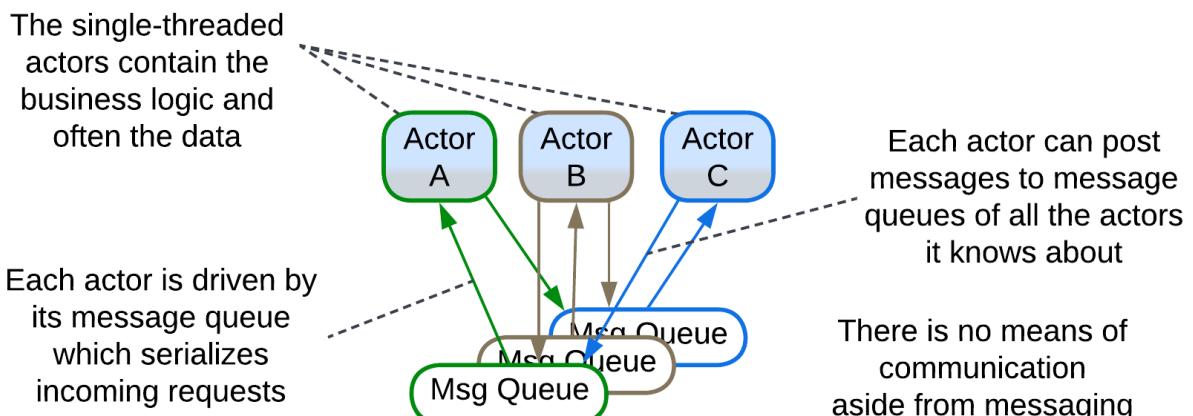
Examples: torrent, onion routing (Tor), blockchain.

Leaf-Spine Architecture, Spine-Leaf Architecture



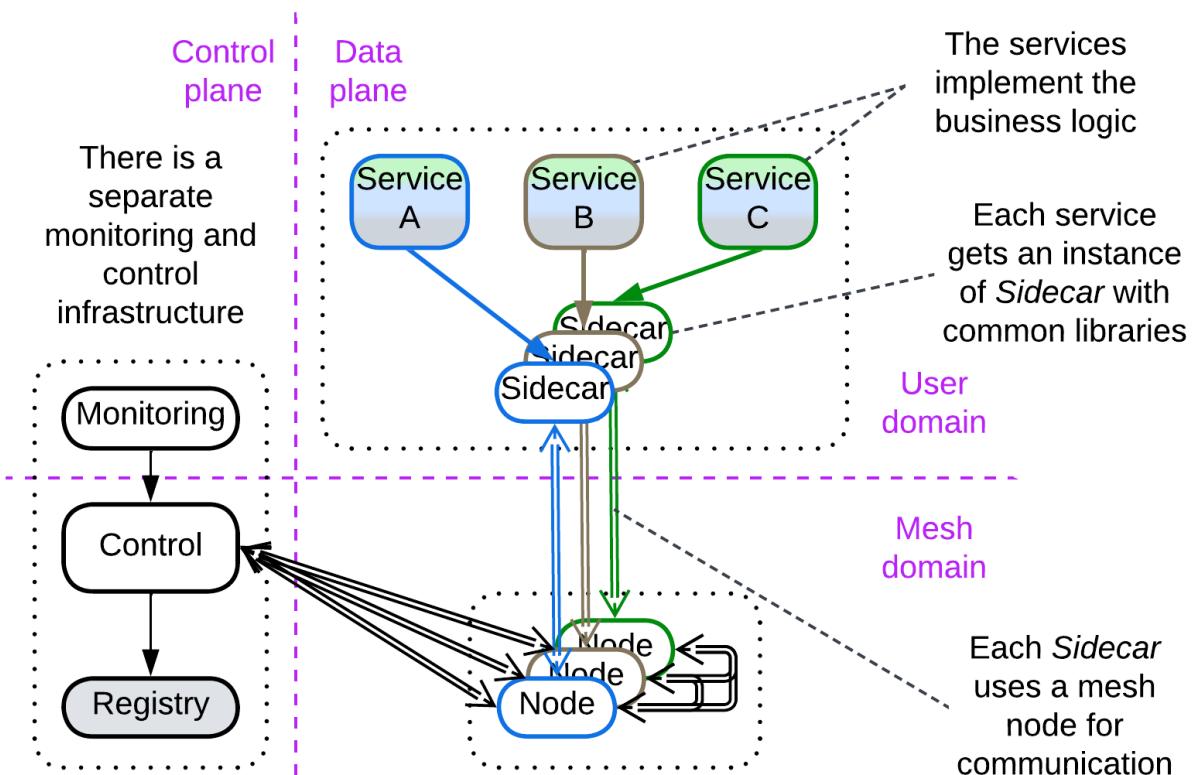
This [datacenter network architecture](#) is a rare example of a *structured fully connected Mesh*. It consists of client-facing (*leaf*) and internal (*spine*) switches. Each *leaf* is connected to every *spine*, allowing for very high bandwidth (by distributing the traffic over multiple routes) that is almost insensitive to failures of individual hardware as there are always many parallel connections.

Actors



A system of *Actors* may be classified as a *fully connected Mesh* with the actors' message queues being the nodes of the *Mesh*. Any actor can post messages to the queue of any other actor it knows about, as all the actors share a virtual namespace or physical address space.

Service Mesh



A [Service Mesh](#) [FSA, MP] is a distributed [Middleware](#) for running [Microservices](#). It is a 2-layer Mesh which contains one or few management nodes (*control plane*) and many user nodes (*data plane*). Each data plane node collocates:

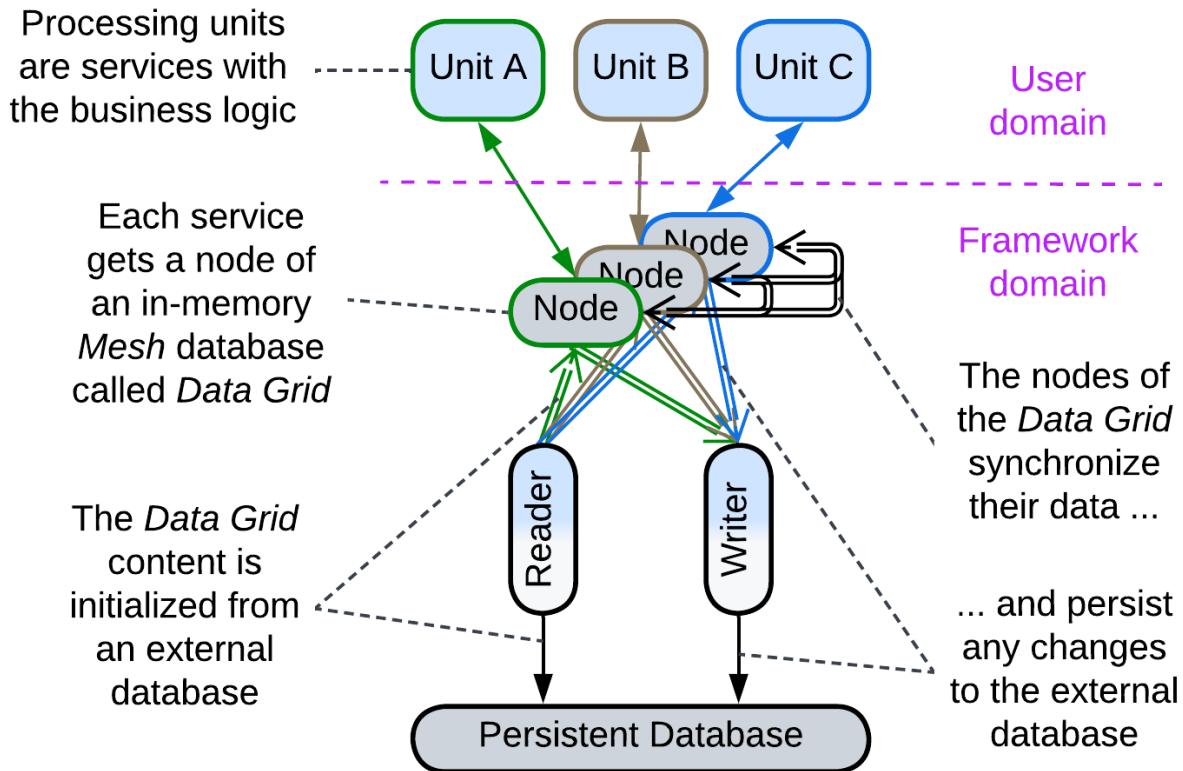
- A *mesh engine node* that deals with connectivity,
- One or more [Sidecars](#) [DDS] ([Proxies](#) where the support of *cross-cutting concerns* – the identical code for use by every service, e.g. logging or encryption – resides),
- A *user application (microservice)* that differs from node to node.

The control plane (re-)starts, updates, scales, and collects statistics from the nodes of the data plane.

Service Mesh addresses some of the weaknesses of naive [Services](#): it provides tools for centralized management and allows for virtual [sharing](#) (through creating physical copies) of libraries to be accessed by all the service instances. It also takes care of scaling and load balancing.

Ready-to-use Service Mesh frameworks are popular with the *Microservices* architecture.

Space-Based Architecture



[Space-Based Architecture](#) [SAP, FSA] is a kind of [Service Mesh](#) with an integrated [Shared Repository](#) (a [tuple space](#) – shared dictionary – called *Data Grid*) and an [Orchestrator](#) (called *Processing Grid*). The user services are called *Processing Units*. They may be identical (making [Shards](#)) or different (resulting in [Services](#)). This architecture is used for:

- Highly scalable systems with relatively small datasets in which case the entire database contents are replicated in the memory of each node. This works around the throughput and latency limits of a normal database.
- Huge datasets, with each node owning a part of the total data. This hacks around the storage capacity and latency limits of a database which may even be kept out of the loop, leaving the *Mesh* as the only data storage.

There are multiple instances of the same data in *Processing Units*. Any change to the data in one unit must propagate to other units. That can be done in several ways:

- Asynchronously, causing conflicts if the same data is changed elsewhere simultaneously.
- Synchronously, waiting for the propagation results and conflict resolution – a kind of distributed transaction which has poor latency.
- The unit takes write ownership of the data before the write. That is not good for latency as well, but it may be a good choice for an evenly distributed load if the *Mesh* engine provides temporary locality of requests, i.e. it forwards requests that touch the same data to the same node.

The choice of the strategy depends on your domain.

The in-memory data in the nodes is usually loaded from a *Persistent Database* on initialization of the system and any change to the data is replicated asynchronously back to the *Persistent Database*, which serves as a means of fault recovery in the unlikely case the entire *Mesh* goes down.

Summary

Mesh is a layer of intercommunicating instances of an infrastructure component that makes a foundation for running custom services in a distributed environment. This architecture is famous for its scalability and fault tolerance but is too complex to implement in-house and may incur performance, administration and development overhead.

Part 6. Analytics

This part is dedicated to analyzing the architectural metapatterns, for if this book's taxonomy of patterns is a step forward from the state of the art, it should bear fruits for us to pick.

I had no time to research every idea collected while the book was being written and its individual chapters published for feedback. A few of those pending topics, which may make additional chapters in the future, are listed below:

- Some architectural patterns (*CQRS*, *Cache*, *Microservices*, etc.) appear under multiple metapatterns. Each individual case makes a story of its own, teaching about both the needs of software systems and uses of metapatterns.
- There are different ways to split a component into subcomponents: [*Layers of Services*](#) differ from [*Layered Services*](#). This should be investigated.
- An architectural quality may depend on the structure of the system. For example, a client request may be split into three subrequests to be processed sequentially or in parallel, depending on the topology (e.g. [*Pipeline*](#), [*Orchestrated Services*](#) or [*Replicas*](#)), thus it is topology which defines latency. We can even draw formulas like:
 - $L = L_1 + L_2 + L_3$ for *Pipeline*,
 - $L = \text{MAX}(L_1, L_2, L_3)$ for *Orchestrated Services* which run in parallel,
 - $L = \text{MIN}(L_1, L_2, L_3)$ for *Replicas* with [*Request Hedging*](#).

Other smaller topics that I was able to look into made the following chapters:

Comparison of architectural patterns

This chapter is a compilation of small sections each of which examines one aspect of the architectural patterns included in this book. It shows the value of having a list of metapatterns to iterate over and analyze.

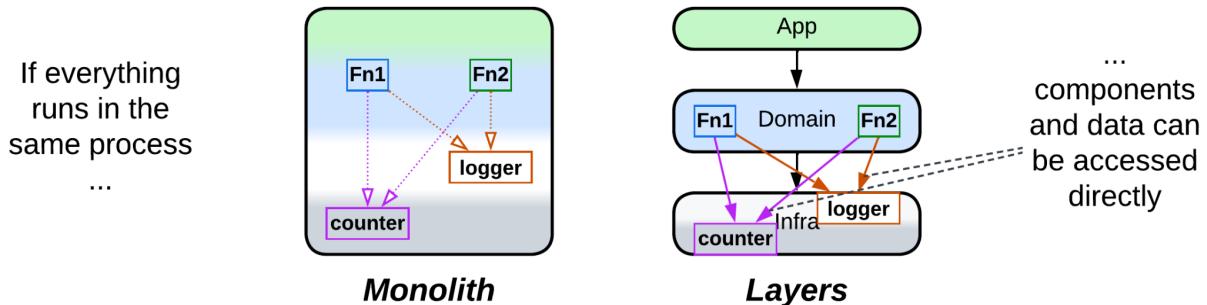
Sharing functionality or data among services

Architectural patterns manifest several ways of sharing functionality or data among their components. Let's consider a basic example: calls to two pieces of business logic need to be logged, while the logger is doing something more complex than mere console prints. The business logic also needs to access a system-wide counter.

Direct call

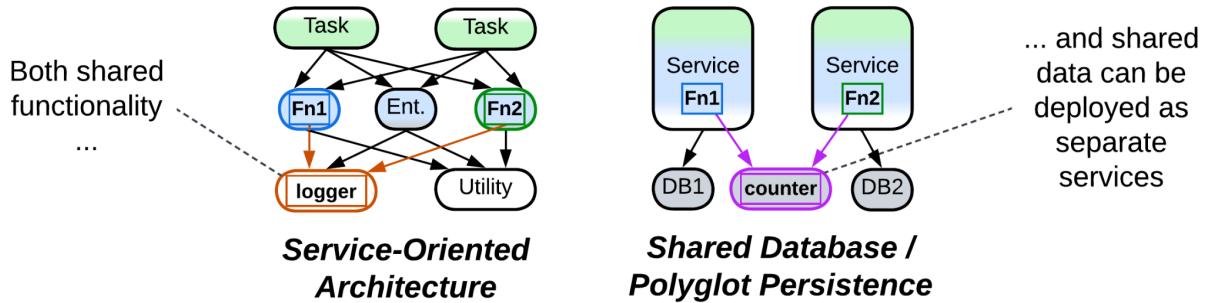
The simplest way to use a shared functionality (aspect) is to call the module which implements it directly. This is possible if the users and the provider of the aspect reside in the same process, as in a [Monolith](#) or module-based (single application) [Layers](#).

Sharing data inside a process is similar, but usually requires some kind of protection, like an [RW lock](#), around it to serialize access from multiple threads.



Make a dedicated service

In a distributed system you can place the functionality or data to share into a separate service to be accessed over the network, yielding [Service-Oriented Architecture](#) for shared utilities or a [Shared Repository / Polyglot Persistence](#) for shared data.

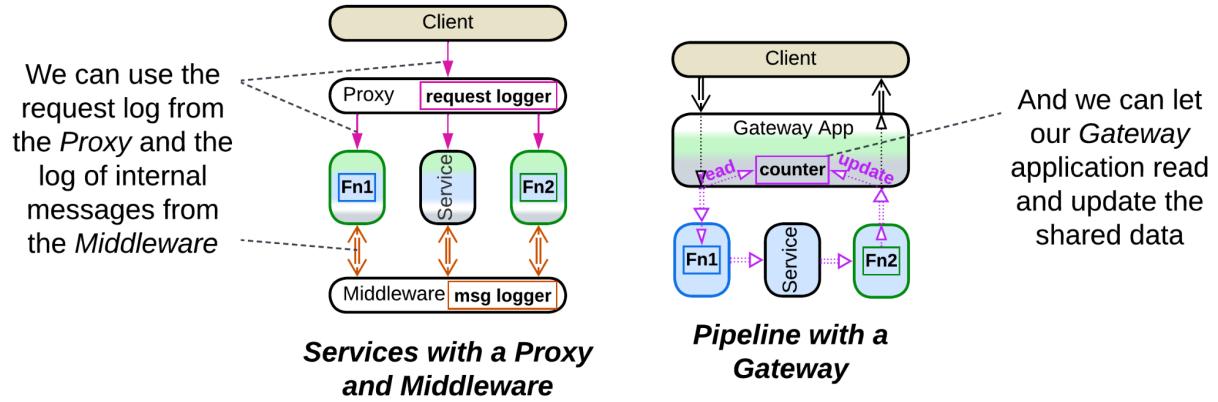


Delegate the aspect

A less obvious solution is [delegating](#) our needs to another layer of the system. To continue our example of logging, a [Proxy](#) may log user requests and a [Middleware](#) –

interservice communication. In many cases one of these generic components is configurable to record all calls to the methods which we need to log – with no changes to the code!

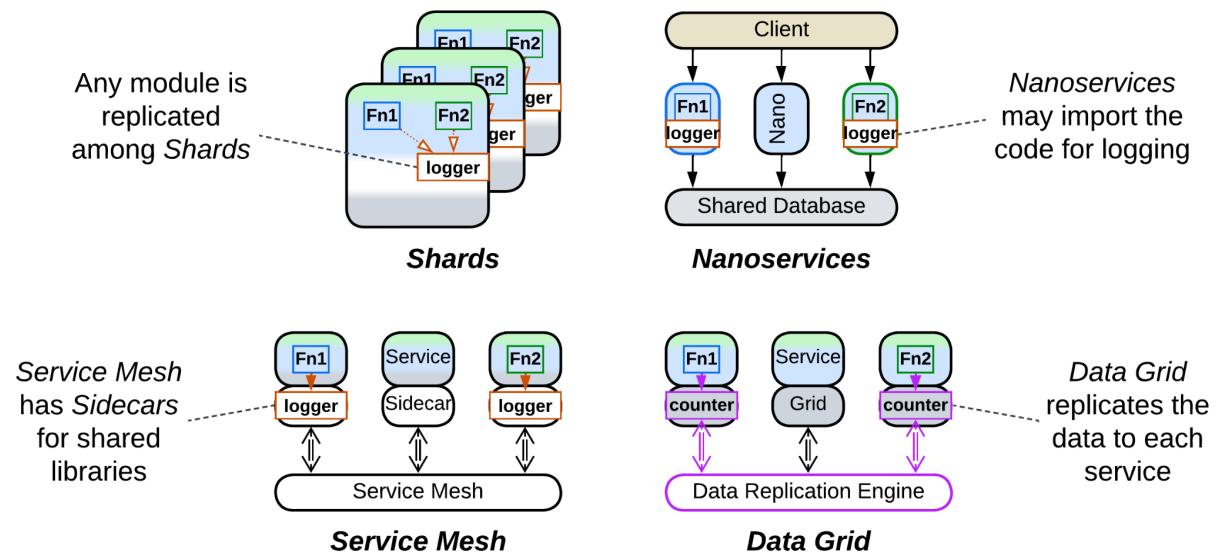
In a similar way a service may [behave as a function](#): receive all the data it needs in an input message and send back all its work as an output – and let the database access remain the responsibility of its caller.



Replicate it

Finally, each user of a component can get its own replica. This is done implicitly in [Shards](#) and explicitly in a [Service Mesh](#) of [Microservices](#) for libraries or [Data Grid](#) of [Space-Based Architecture](#) for data.

Another case of replication is importing the same code in multiple services, which happens in [single-layer Nanoservices](#).



Summary

There are four basic ways to share functionality or data in a system:

- Deploy everything together – messy but fast and simple.
- Place the component in question into a shared service to be accessed over the network – slow and less reliable.
- Let another layer of the system both implement and use the needed function on your behalf – easy but generic, thus it may not always fit your code's needs.
- Make a copy of the component for each of its users – fast, reliable, but the copies are hard to keep in sync.

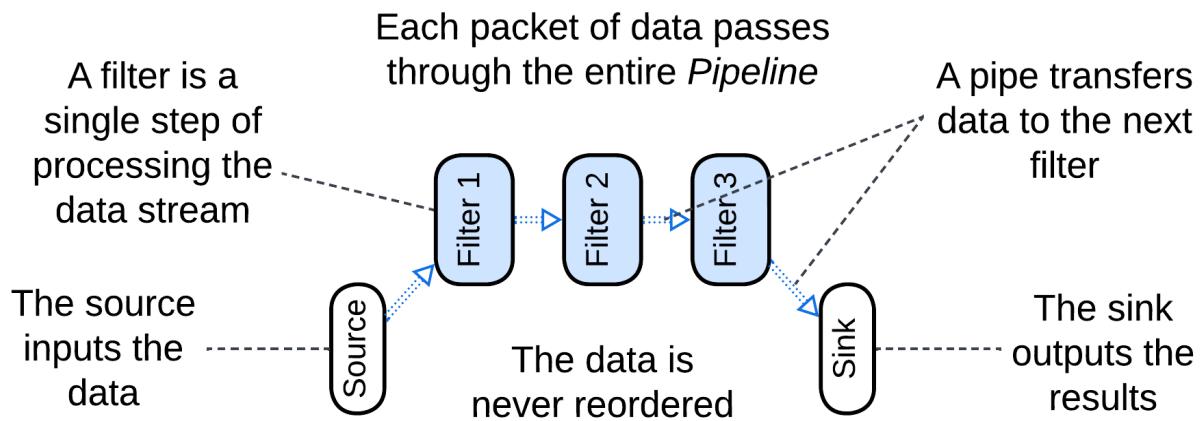
Pipelines in architectural patterns

Several architectural patterns involve a unidirectional data flow – a [pipeline](#). Strictly speaking, every data packet in a pipeline should:

- Move through the system over the same *route* with no loops.
- Be of the same *type*, making a *data stream*.
- Retain its *identity* on the way.
- Retain *temporal order* – the sequence of packets remains the same over the entire pipeline.

Staying true to all of these points makes *Pipes and Filters* – one of the oldest known architectures. Yet there are other architectures that discard one or more of the conditions:

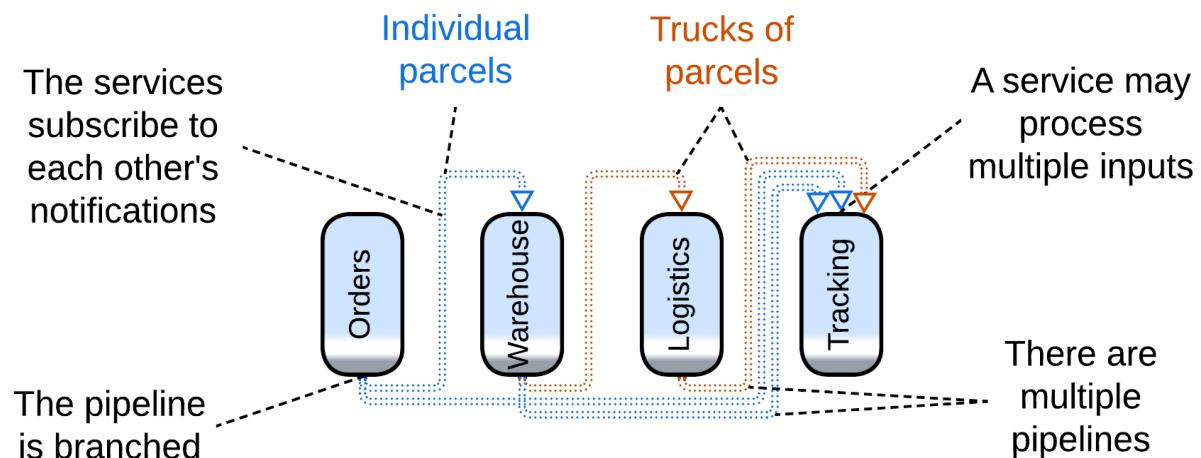
Pipes and Filters



[Pipes and Filters \[POSA1\]](#) is about stepwise [processing of a data stream](#). Each piece of data (a video frame, a line of text or a database record) passes through the entire system.

This architecture is easy to build and has a wide range of applications, from hardware to data analytics. Though each pipeline is specialized for a single use case, a new one can often be built of the same set of generic components – this skill is mastered by Linux admins through their use of shell scripts.

Choreographed Event-Driven Architecture

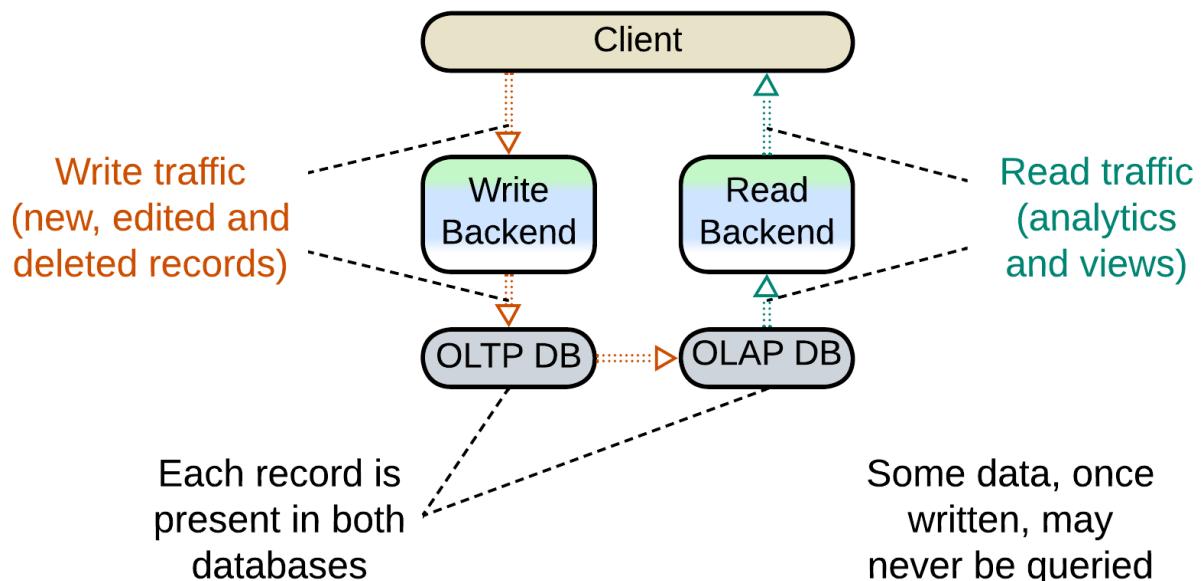


Relaxing the *type* and loosening the *identity* clauses opens the way to [Choreographed Event-Driven Architecture](#) [SAP, FSA], in which a service publishes notifications about anything it does that may be of interest to other services. In such a system:

- There are multiple types of events going in different directions, like if several branched pipelines were built over the same set of services.
- A service may aggregate multiple incoming events to publish a single, seemingly unrelated, event later when some condition is met. For example, a warehouse delivery collects individual orders till it gets a truckload of them or until the evening comes and no new orders are accepted.

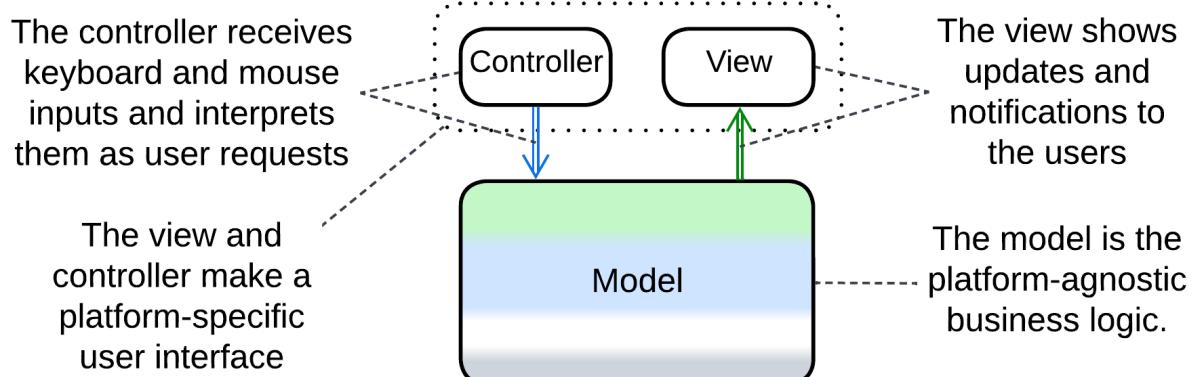
This architecture covers way more complex use cases than [Pipes and Filters](#) as multiple pipelines are present in the system and because processing an event is allowed to have loosely related consequences (as with the parcel and truck).

Command Query Responsibility Segregation (CQRS)



When data from events is stored for a future use (as with the aggregation above), the type and *temporal order* are ignored but data *identity* may be retained. A [CQRS-based system \[MP\]](#) separates paths for write (*command*) and read (*query*) requests, making a kind of data processing pipeline with the database in the middle, which stores events for an indeterminate amount of time. It is the database that reshuffles the order of events, as a record it stores may be queried at any time, maybe in a year from its addition – or never at all.

Model-View-Controller (MVC)



[Model-View-Controller](#) [POSA1] completely neglects the *type* and *identity* limitations. It is a coarse-grained pattern where the input source produces many kinds of events that go to the main module which does something and outputs another stream of events of no obvious relation to the input. A mouse click does not necessarily result in a screen redraw, while a redraw may happen on timer with no user actions. In fact, the pattern conjoins two different short pipelines.

Summary

There are four architectures with unidirectional data flow, which is characteristic of [pipelines](#):

- [Pipes and Filters](#),
- [Choreographed Event-Driven Architecture \(EDA\)](#),
- [Command \(and\) Query Responsibility Segregation \(CQRS\)](#),
- [Model-View-Controller \(MVC\)](#).

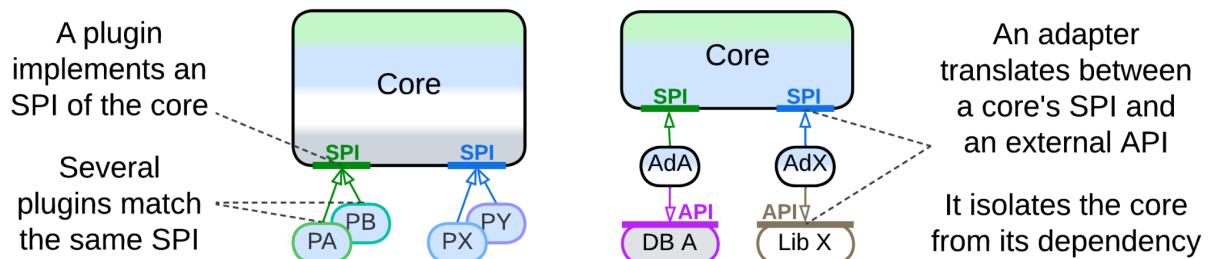
The first two, being true pipelines, are built around data processing and transformation, while for the others it is just an aspect of implementation – their separation of input and output yields pairs of streams.

Dependency inversion in architectural patterns

I am no fan of [SOLID](#) – to the extent of being unable to remember what those five letters mean – thus I was really surprised to notice that one of its principles, namely [dependency inversion](#), is quite common with architectural patterns, which means that it is way more generic than *OOP* it is promoted for.

Let's see how dependency inversion is used on system level.

Patterns that build around it



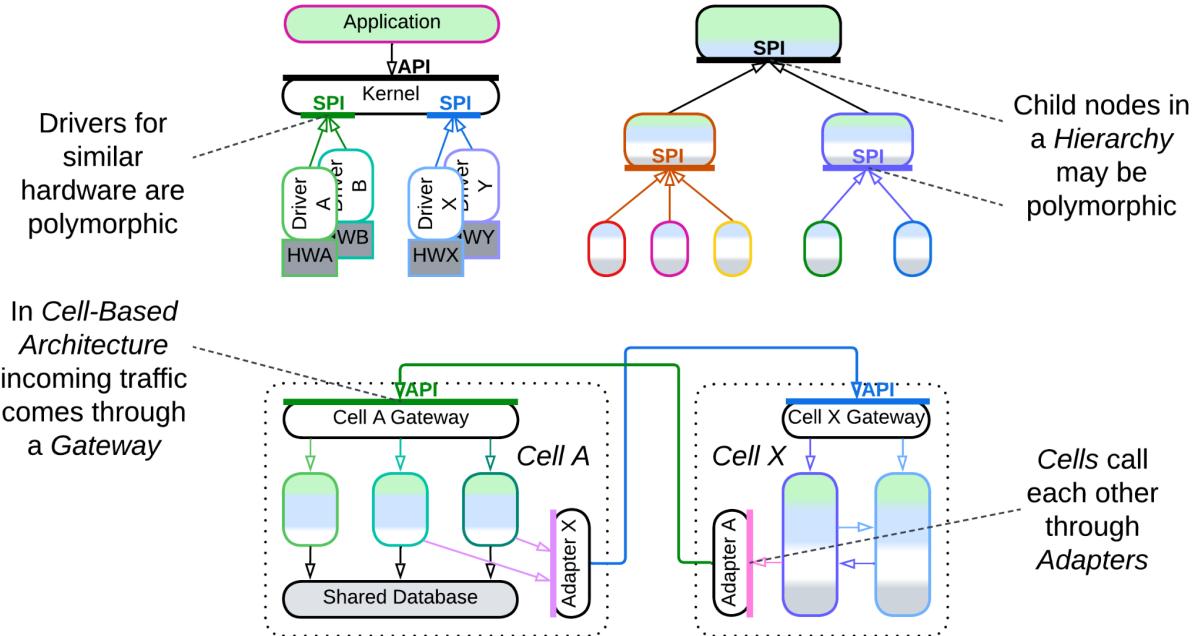
Both [Plugins](#) and the derived [Hexagonal Architecture](#) rely on dependency inversion for the same reason – to protect the *core*, which contains the bulk of the code, from variability in the external components it uses. The *core* operates interfaces ([SPIs](#)) which it defines so that it may not care whatever is behind an interface.

It is the nature of the polymorphic components that distinguishes the patterns:

- [Plugins](#) allow for small pieces of code, typically contributed by outside developers, to provide customizable parts of the system's algorithms and decision making. Oftentimes the core team has no idea of how many diverse plugins will be written for their product.
- [Hexagonal Architecture](#) is about breaking dependency of the core on external libraries or services by employing [adapters](#). Each adapter depends both on the core's SPI and on the API of the component which it adapts. As interfaces and contracts vary among vendors and even versions of software, which we want to be

interchangeable, we need adapters to wrap the external components to make them look identical to our core. Besides, [stub or mock](#) adapters help develop and test the core in isolation.

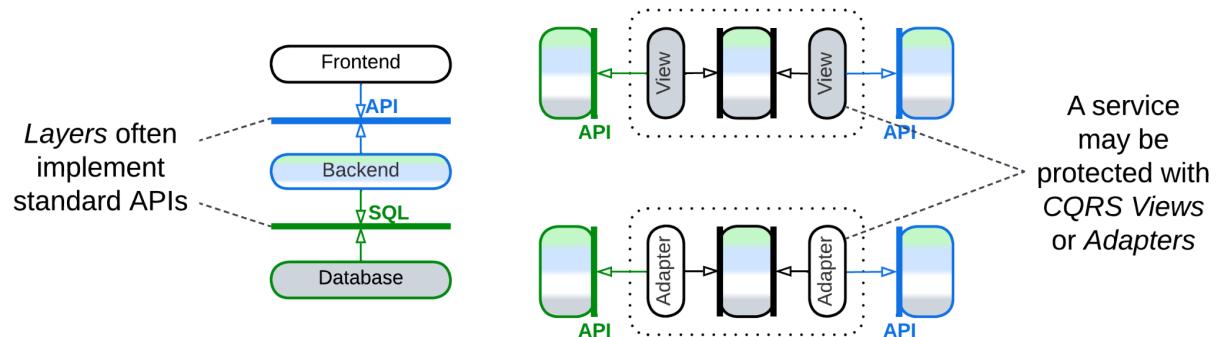
Patterns that often rely on it



A few more metapatterns tend to use this approach to earn its benefits, even though dependency inversion is not among their integral features:

- [Microkernel](#), yet another metapattern derived from [Plugins](#), distributes resources of *providers* among *consumers*. Polymorphism is crucial for some of its variants, including [Operating System](#), but may rarely benefit others, such as [Software Framework](#).
- [Top-Down Hierarchy](#) distributes responsibility over a tree of components. If the nodes of the tree are polymorphic, they are easier to operate, and there is dependency inversion. However, in practice, a parent node may often be strongly coupled to the types of its children and access them directly.
- In another kind of [Hierarchy](#), namely [Cell-Based Architecture](#) (aka Services of Services), each [Cell](#) may employ a [Cell Gateway](#) and outbound [Adapters](#) to isolate its business logic from the environment – just like [Hexagonal Architecture](#) does for its monolithic core.

Patterns that may use it



Finally, two basic architectures, [Layers](#) and [Services](#), may resort to something similar to dependency inversion to decouple their constituents:

- We often see a higher layer to depend on and a lower layer to implement a standardized interface, like POSIX or SQL, to achieve *interoperability* with other implementations (which is yet another wording for polymorphism).
- A service may follow the concept of [Hexagonal Architecture](#) by using an [Anti-
Corruption Layer](#) [DDD] or [CQRS Views](#) [MP] as [Adapters](#) that protect it from changes in other system components.

Summary

Many architectural patterns employ dependency inversion by adding:

- an *interface* to enable polymorphism of their lower-level components or
- [Adapters](#) to protect a component from changes in its dependencies.

The two approaches apply in different circumstances:

- If you can enforce your rules of the game on the suppliers of the external components, you merely *define an SPI*, and expect the suppliers to implement and obey it.
- If the suppliers are independent and it is your side that adapts to their rules, you should *add Adapters* to translate between your lovely SPI and their whimsical APIs.

Indirection in commands and queries

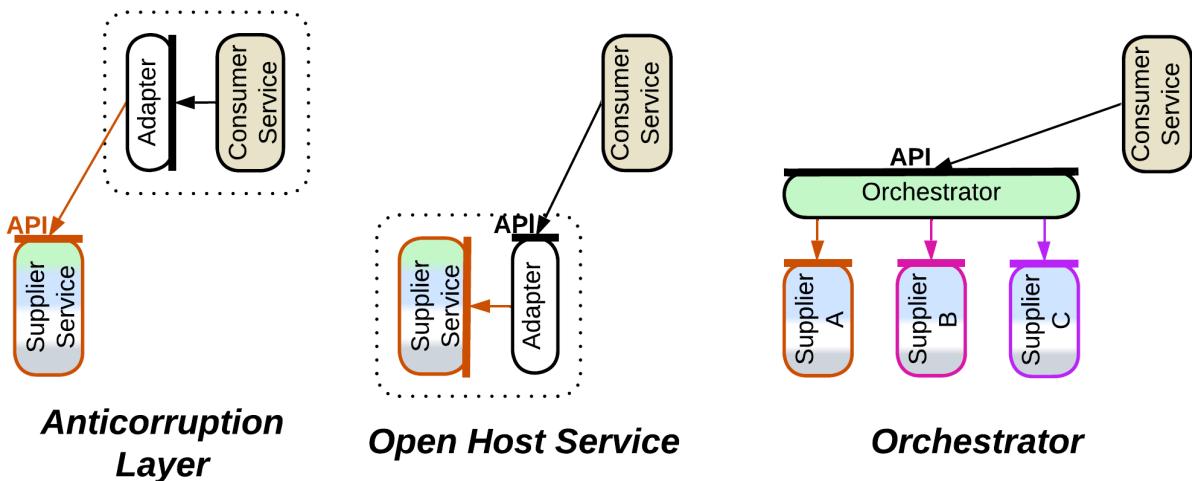
We can solve any problem by introducing an extra level of indirection – states the [old adage](#). We will not explain how this rule drives [deep learning](#), at least for now. Instead, let's concentrate our effort on indirection in communication between services.

Each component operates its own *domain model* [DDD] which translates into objects and/or procedures convenient for use in the component's subdomain. However, should a system cover multiple subdomains, the best models for its parts to operate start to mismatch. Furthermore, they are likely to diverge progressively over time as requirements heap up and [the project matures](#).

If we want for each module or service to continue with a model that fits its needs, we have to protect it from the influence of models of other components by employing indirection – a translator – between them.

In a system of subdomain-dedicated [Services](#) a service may need to operate entities that are defined in another service's subdomain. For example, the financial and recruiting departments' software operates employees, but the employee data which each department needs is different. Moreover, it also differs from the employee records in the HR department which is responsible for adding, editing, and discarding the employees. We don't want our accountants to spend their nights seeking the correlation between salaries, birthday horoscopes from HRs, and MBTI tests from the recruiters.

Command (OLTP) systems

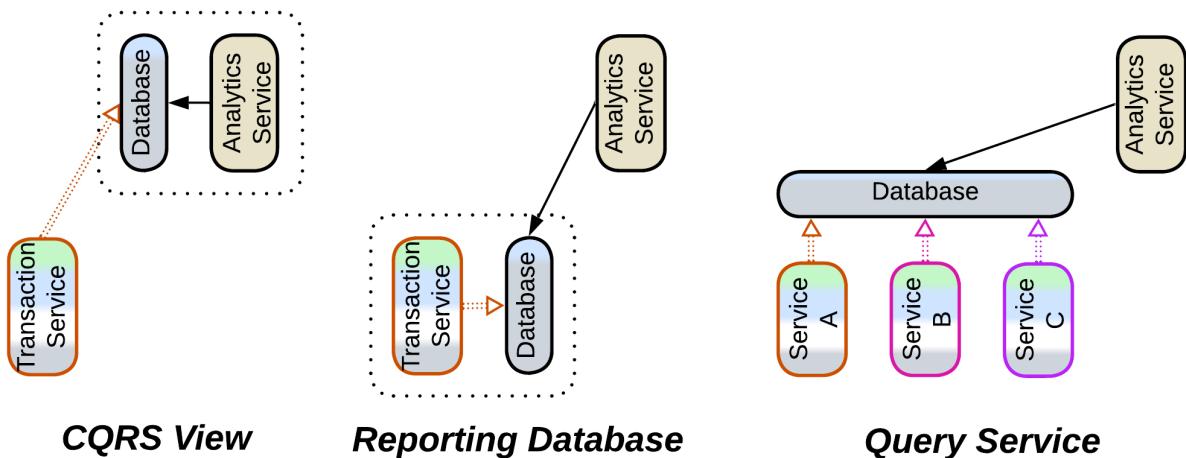


More often than not our system consists of services that command each other: via [RPCs](#), requests or even notifications – no matter how, one component makes a call to action which other(s) should obey.

In such a case we employ an [Adapter](#) between two services or an [Orchestrator](#) when cooperation of several services is needed to execute our command:

- An [Anticorruption Layer](#) [DDD] is an *Adapter* on the dependent service's side: as soon as we call another service, we start depending on its interface, while it is in our interest to isolate ourselves from its peculiarities and possible future changes. Thus we should better write and maintain a component to translate the foreign interface, defined in terms of the foreign domain model, into terms convenient for use with our code. Even if we subscribe to notifications, we may also want to have an *Adapter* to transform their payload.
- An [Open Host Service](#) [DDD] resides on the other side of the connection – it is an *Adapter* that a service provider installs to hide the implementation details of its service from its users. It will typically translate from the provider's domain model into a more generic (subdomain-agnostic) interface suitable for use by services that implement other subdomains.
- An [Orchestrator](#) (which can be an [API Composer](#) [MP], [Process Manager](#) [EIP, LDDD], or [Saga Orchestrator](#) [MP]) spreads a command to multiple services, waits for each of them to execute their part, and cleans up after possible failures. It tends to be more complex than other translators because of the coordination logic involved.

Query (OLAP) systems



There is often another aspect of communication in a system, namely, information collection and analysis. And it runs into a different set of issues which cannot be helped by mere interface translation.

Each service operates and stores data in its own format and schema which matches its *domain model*, as discussed above. When another service needs to analyze the foreign data according to its own domain model, it encounters the fact that the foreign format(s) and schema(s) don't allow for efficient processing – in the worst case it would have to read and re-process the entire foreign service's dataset to execute a query.

The solution employs an intermediate database as a translator from the provider's to consumer's preferred data access mode, format, and schema:

- A [CQRS View \[MP\]](#) resides in the data consumer and aggregates the stream of changes published by the data provider. This way the consumer can know whatever it needs about the state of the provider without making an interservice call.
- [Data Mesh \[LDDD\]](#) is about each service exposing a general-use public interface for streaming and/or querying its data. Maintaining one often requires the service to set up an internal [Reporting Database](#).
- A [Query Service \[MP\]](#) aggregates streams from multiple services to collect their data together, making it available for efficient queries (joins).

Summary

We see that though command-dominated (*operational* or *transactional*) and query-dominated (*analytical*) systems differ in their problems, the architectural solutions which they employ to decouple their component services match perfectly:

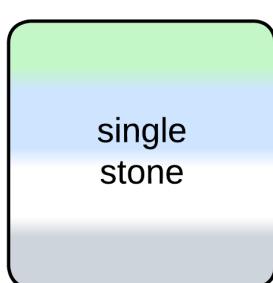
- [Anticorruption Layer](#) or [CQRS View](#) is used on the consumer's side,
- [Open Host Service](#) or [Data Mesh's Reporting Database](#) is on the provider's side,
- [Orchestrator](#) or [Query Service](#) coordinates multiple providers.

Which shows that the principles of software architecture are deeper than the [CQRS](#) dichotomy.

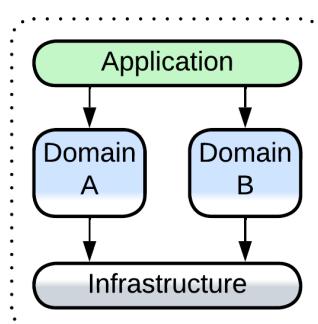
Ambiguous patterns

We've seen a single pattern come under many names, as happens with [Orchestrator](#), and also one name used for multiple topologies, as with [Services](#), which may [orchestrate each other](#), make a [Pipeline](#), or be components of the [Service-Oriented Architecture](#) (SOA). On top of that, there are several pattern names that are often believed to be unambiguous while each of them sees conflicting definitions in books or over the web (thanks to [Semantic Diffusion](#) or independent coining of the term by multiple authors). Let's explore the last kind, which is the most dangerous both for your understanding of other people and for your time wasted in arguments.

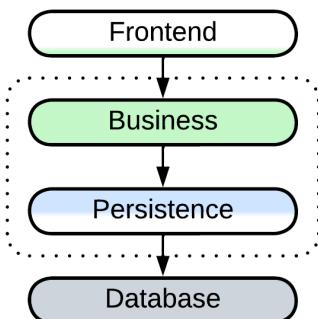
Monolith



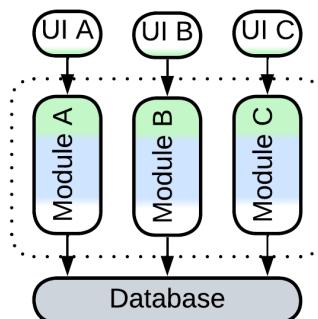
Ye Olde Monolith



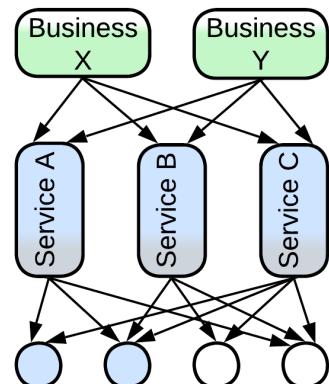
New Gen Monolith



Layered Monolith



Modular Monolith



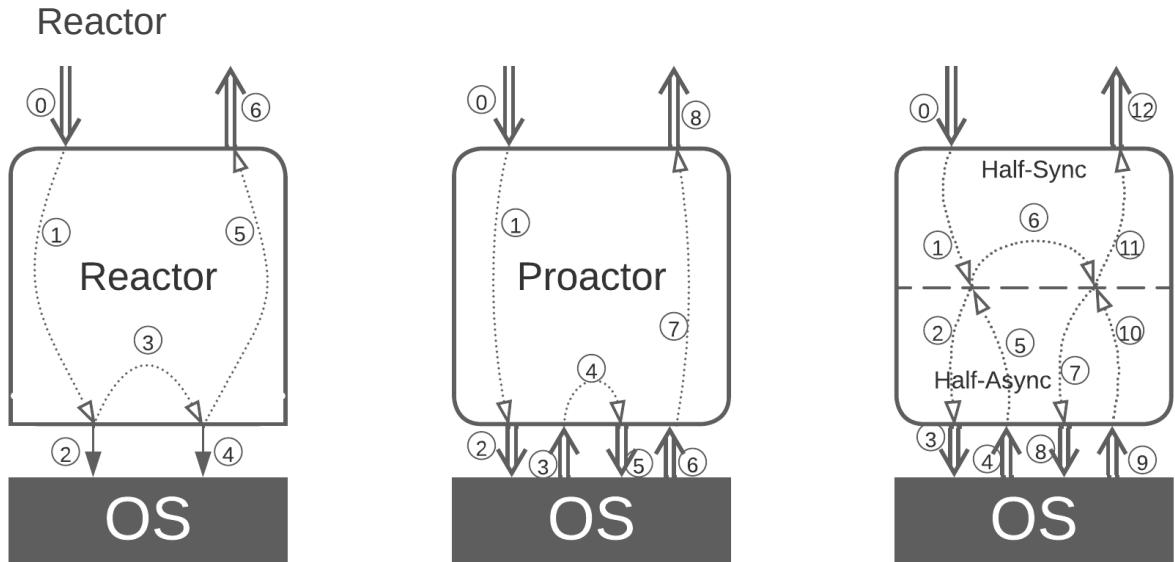
Distributed Monolith

The old books, namely [\[GoF\]](#) and [\[POSA1\]](#), described a *tightly coupled unstructured system*, where anything depends on everything, as *monolithic*, which matched the meaning of the word in Latin – “single stone”.

Then something evil happened – I believe that the proponents of [SOA](#), backed by the hype and money they had earned from corporations, started labeling any *non-distributed* system as *monolithic*, obviously to contrast the negative connotation of the word to their own most progressive design.

It took only a decade for the karma to strike back – when the new generation behind [Microservices](#) redefined *monolithic* as a *single unit of deployment* – to call the now obsolescent SOA systems [Distributed Monoliths](#) [MP] because their services used to grow so coupled that they had to be deployed together.

The novel misnomers, [Layered Monolith](#) [FSA] and [Modular Monolith](#) [FSA], which denote an application partitioned by abstractness or subdomain, correspondingly, add to the confusion.

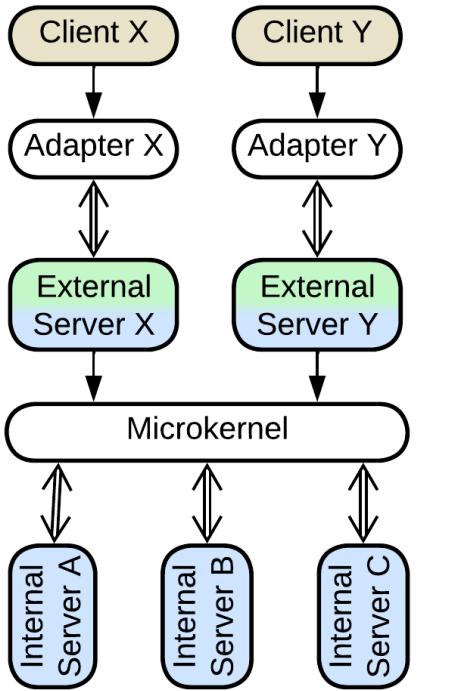


People [tend to call](#) any event-driven service a *Reactor*. In fact, there are three patterns that describe threading models for an event-handling system:

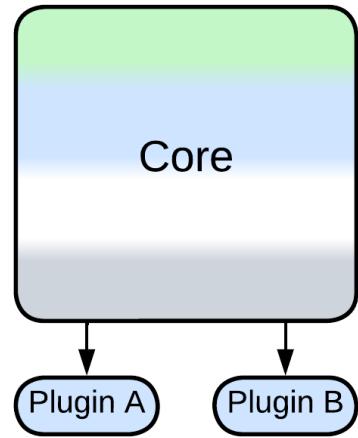
- A [Reactor](#) [POSA2] runs each task in a dedicated thread and blocks it on any calls outside of the component. That allows for normal *imperative programming* but is resource-consuming and not very responsive or flexible.
- A [Proactor](#) [POSA2] relies on a single thread to serve all the system's tasks in an interleaved manner, just like an OS uses a CPU core to run multiple processes. The resulting non-blocking code is fragmented (thus known as *callback hell*) but it can address any incoming event immediately. This suits [real-time control systems](#).
- [Half-Sync/Half-Async](#) [POSA2] is what we know better as *coroutines* or *fibers* – there are multiple *Reactor*-like lightweight threads that block on a *Proactor*-like engine which translates between synchronous calls from the user code and asynchronous system events.

In most cases we'll hear of *Proactor* being called *Reactor* – probably because *Reactor* was historically the first and the simplest of the three patterns, and it is similar in name to *reactive programming* found in *Proactor*.

Microkernel



The Original Microkernel



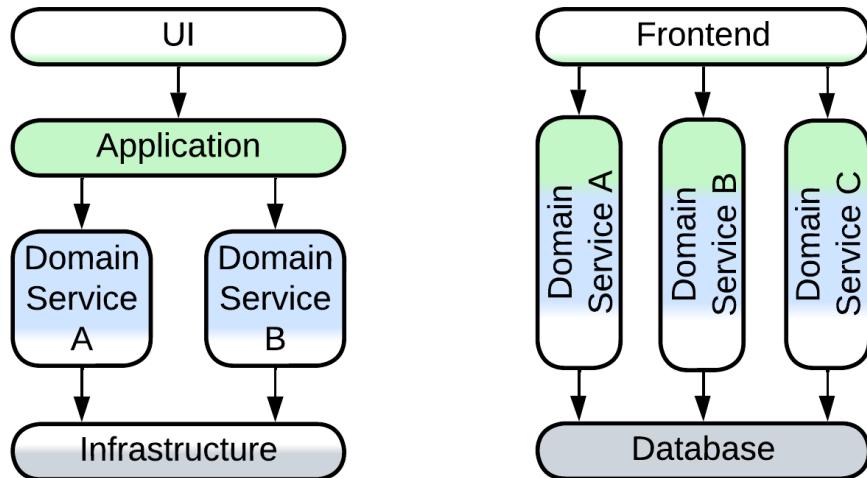
Microkernel aka Plugins

Microkernel is another notable case. The confusion over it goes all the way back to [POSA1] which used [operating systems](#) as examples of [Plugin Architecture](#). I believe that it was a mismatch:

- An operating system is mainly about sharing the resources of producers among consumers, where both the producers and consumers may be written by external teams. The *kernel* itself does not feature much logic – its role is to connect the other components together.
- *Plugins*, on the other hand, extend or modify the business logic of the *core* – which alone is the reason for the system to exist and is in no way “*micro-*” as it got the bulk of the system’s code. In many such systems *plugins* are utterly optional – which cannot be said of OS *drivers*.

Thus, here we have two architectural patterns of arguably similar structure ([Microkernel/Plugins](#) of [SAP, FSA] omit 3 of 5 components of the original [Microkernel](#) of [POSA1, POSA4]) but very different intent and action known under the same name.

Domain Services



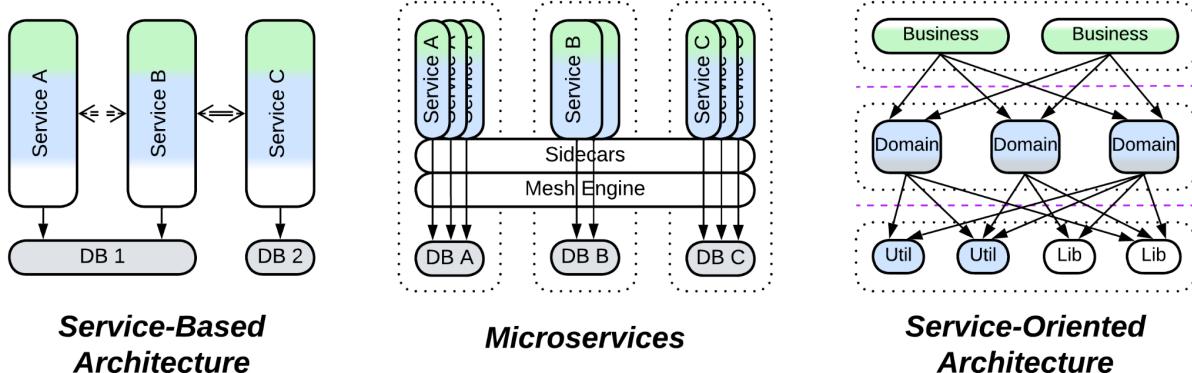
Domain Services of DDD

Domain Services of FSA

I was told that [Domain Services](#) of [FSA] are an incorrect term – because a *domain service* is always limited to the [domain layer](#) of [DDD] while those of [FSA] also cover the *application* and, maybe, *infrastructure*.

I believe that both definitions are technically correct, if the difference in the meaning of *domain* is accounted for. In [FSA] *domain* is almost synonymous with a *bounded context* of [DDD], while [DDD] more often uses that word for the name of its middle layer which contains *business rules*.

Service-Based Architecture



Service-Based Architecture

Microservices

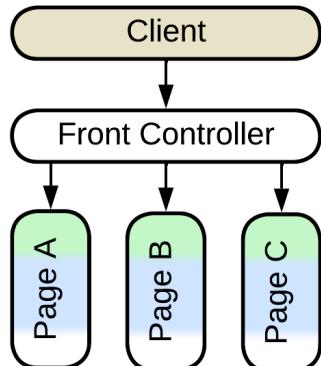
Service-Oriented Architecture

[DEDS] calls anything service-based a *service-based architecture*.

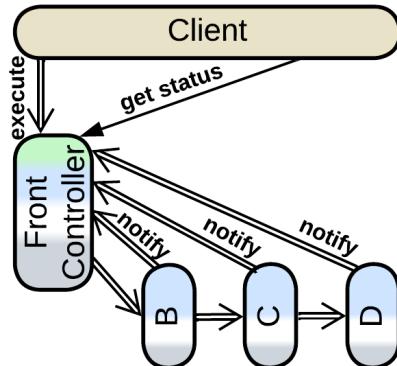
[FSA] differentiates [Microservices](#) and [Service-Oriented Architecture](#), leaving whatever remains (large [subdomain-scale services](#)) under the name of [Service-Based Architecture](#).

Both definitions are technically correct. One is wider than the other.

Front Controller



**Front Controller
of PEAA**

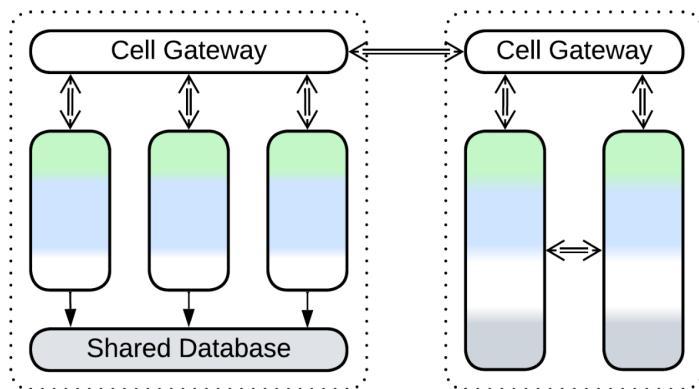


**Front Controller
of SAHP**

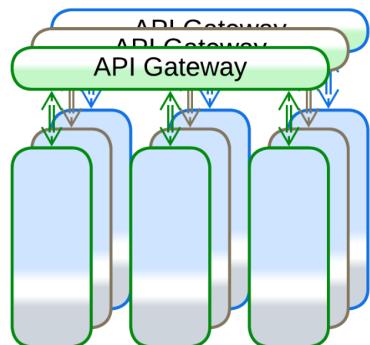
[PEAA] defines [Front Controller](#) as an [MVC](#) derivative for backend programming. In this pattern multiple web pages share a request processing component which turns the incoming requests into commands and forwards them to appropriate page classes.

The definition from [SAHP] is much more interesting – it describes an [Choreographed Event-Driven Architecture](#) with a [Query Service](#) embedded in the first (client-facing) service. The [Front Controller](#) subscribes to notifications from downstream services to know the status of every request it has passed to the [pipeline](#).

Cells



WSO2 Cells

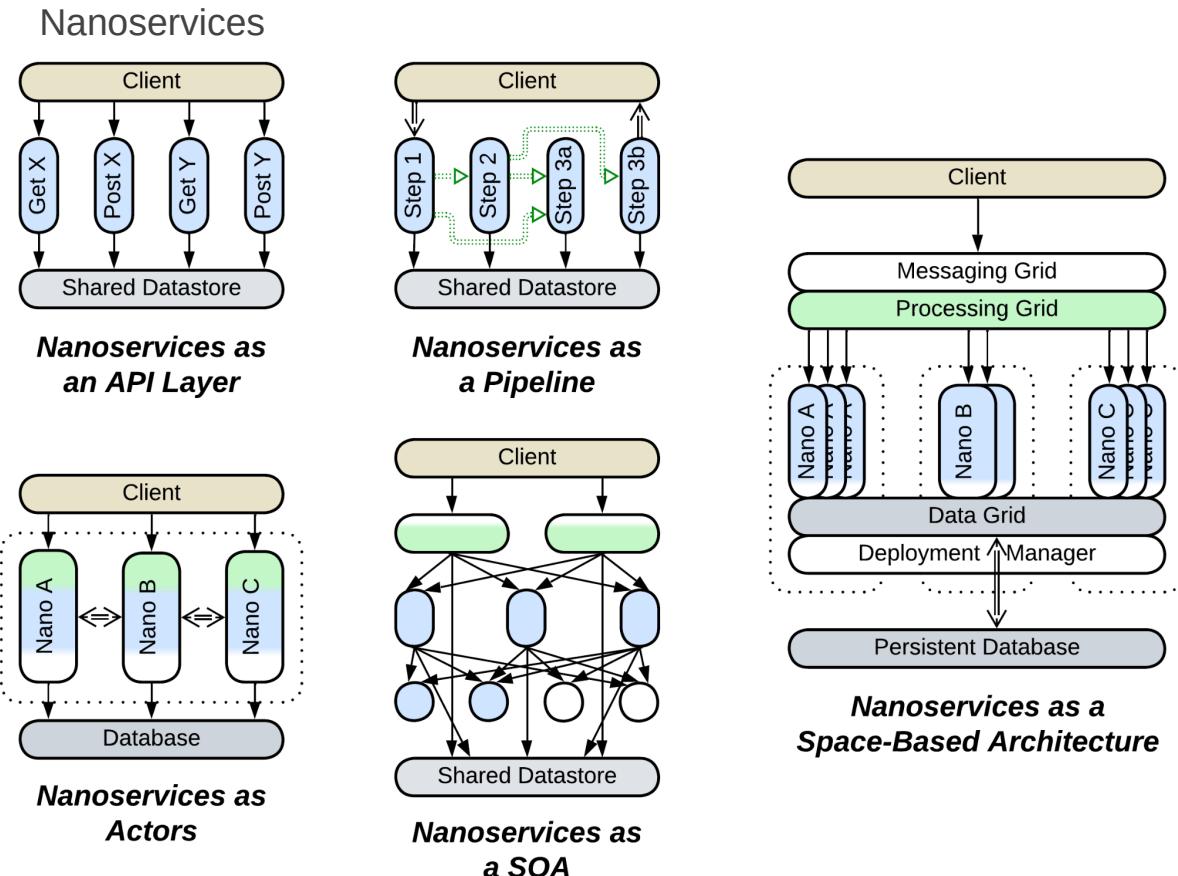


Amazon Cells

The fresh *Cell-Based Architecture* also has multiple definitions.

- WSO2 [wrote](#) about a [Cell](#) as a group of services which is encapsulated from the remaining system by a [Gateway](#) (for incoming traffic) and sometimes *Adapters* (for outgoing traffic) and often uses a dedicated [Middleware](#) – causing each *Cell*, though internally distributed, to be treated by other components as a single service. This makes designing and managing a large system a bit simpler by introducing a [hierarchy](#).
- Amazon [promotes](#) its [Cells](#) as [Shards](#) of the whole system which run in multiple geographic regions. That grants fault tolerance and improves performance as each client has an instance of the system deployed to a nearby datacenter, but it does not have much impact on organization and complexity of the code.

This case looks like Amazon's hijacking and redefining a popular emerging technology, though I may be wrong about that as I did not investigate the history of the term.



The *Nanoservices* pattern is another emerging technology and it seems to have never been strictly defined. Most sources agree that a [nanoservice](#) is a cloud-based function ([FaaS](#)), similar to a service with a single API method but, just as with the old good [services](#), they differ in the ways they use the novel technology:

- Diego Zanon in *Building Serverless Web Applications* proposes a [single layer of nanoservices](#), each implementing a method of the system's public API, to be used as a thin backend.
- [Here](#) we have nanoservices [built into a Pipeline](#), similar to [Choreographed Event-Driven Architecture \[FSA\]](#).
- [Another article](#) proposes to [\(re\)use them](#) in [SOA](#) style.

Moreover, there are a couple of sources that call a nanoservice something totally different:

- [There is a concept](#) of nanoservice as a module that can run both as a separate service and as a part of a binary – allowing for a team to choose if they want their system to run as a single process or become distributed. *Nano-* is because an in-process module is more lightweight than a [microservice](#). This idea resembles [Modular Monolith \[FSA\]](#) and [Actor Frameworks](#).
- And [here](#) we have something akin to [Space-Based Architecture](#) but it is also called *Nanoservices* – as the proposed framework makes new components so easy to create that programmers tend to write many smaller *nanoservices* instead of a single [microservice](#).

In my opinion, the disarray happened because the notion of *making smaller microservices* got hyped but was never adopted widely enough to become an industry standard, therefore everybody follows their own vision about what *smaller* means.

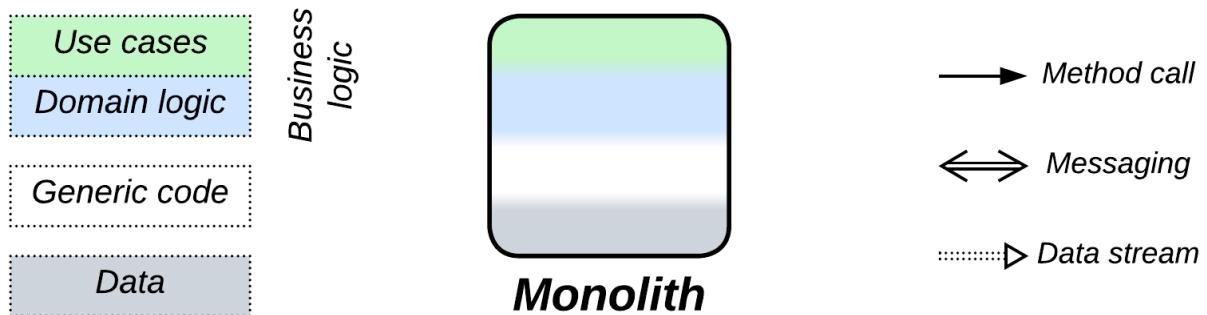
Summary

A few names of architectural patterns cause confusion as the meaning of each of them changes from source to source. This book aims at identifying such issues and building a cohesive understanding of software and system architecture, similar to the *ubiquitous language* of [[DDD](#)].

Architecture and product life cycle

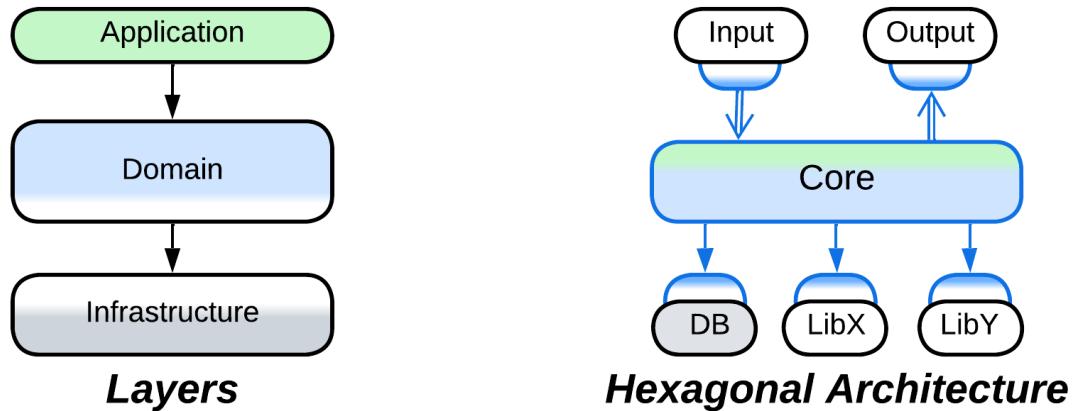
In my practice, a product's architecture changes over its lifetime. For a R&D, when there is nobody with relevant experience on the team, it starts small, gradually gains flexibility through fragmentation, grows and restructures itself according to the ever-changing domain knowledge and business requirements, then it solidifies as the project matures and dies to performance optimizations and loss of experience as main programmers leave. In more mundane projects the first stages may be omitted, as little research needs to be done, and oftentimes a project is canceled way before its architecture succumbs under its own weight. Anyway, let's observe the full life cycle.

Infancy (proof of concept) – Monolith



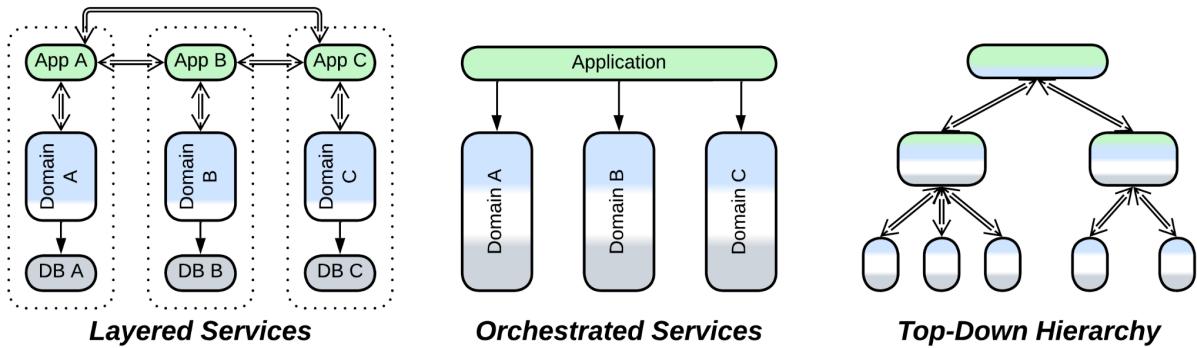
A project in an unknown domain starts humble and small, likely as a proof of concept. You need to write quickly to check your ideas about how the domain works without investing much time – as you may oftentimes be wrong here or there, making you rethink and rewrite.

Childhood (prototype) – Layers



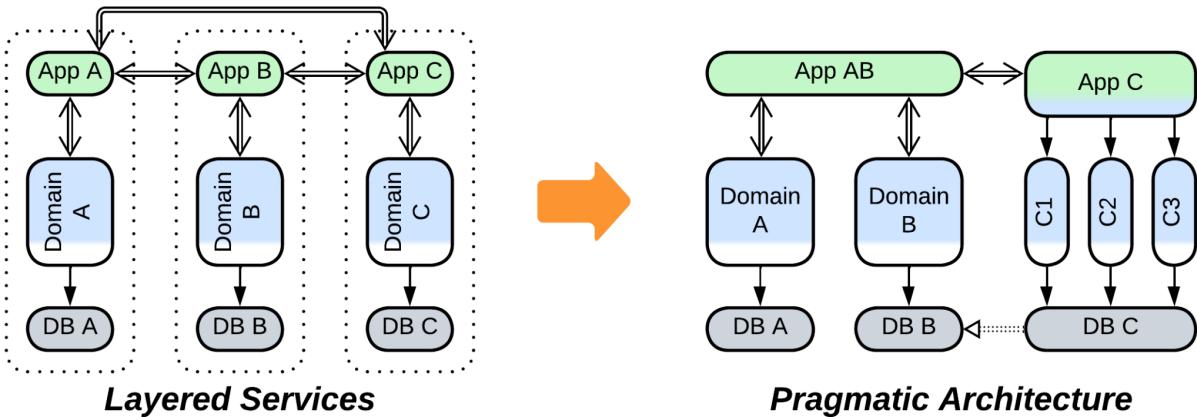
When you have the thing working, you may start reflecting on the rules and the code you wrote. What belongs where, what can be subject to change, which tests will you need? At this point you clearly see the levels of abstractness: the high-level *application* (integration, orchestration) logic, the lower-level *domain* (business) rules, and the generic *infrastructure* [DDD]. Now that you know better the whats and the hows, you divide the code (either old or rewritten from scratch) into [Layers](#) or [Hexagonal Architecture](#) to make it both structured and flexible, still without heavy development overhead caused by interfaces between subdomains.

Youth (development of features) – fragmented architectures



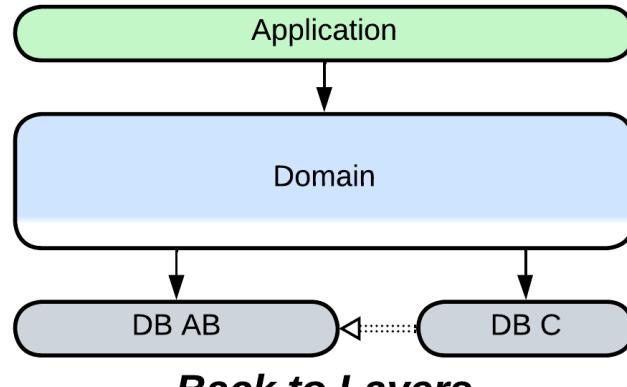
As you acquire domain experience, you start discerning subdomains (or *bounded contexts* [DDD]) and isolating them to reduce the [complexity](#) of your code. The layered structure turns into a system of subdomain-dedicated components: [modules](#), [services](#), [device drivers](#) – whatever you used to name them throughout your career. The actual architecture follows the structure of the domain, with [Layered Services](#), [Orchestrated Services](#), and [Top-Down Hierarchy](#) among common examples. The fragmentation of the system enables development by multiple teams with diverse technologies and styles, reduces ripple effect of changes, and helps testability. However, use cases for the system as a whole become harder to understand and fix – if only because they traverse the parts of the code owned by multiple teams – which is not extremely bad given you have enough manpower to do the work.

Adulthood (production) – ad-hoc composition



As the product enters the market, its development tends to slow down with more attention given to corner cases and user experience. Some (often the most active) people are going to get bored and leave the project, while your understanding of the domain changes again based on user experience and real-life business needs [DDD]. You may find that some of the components which you have designed as independent become strongly coupled, and you are lucky if they are small enough to be merged together – this is where the fragmentation from the previous stage pays off. Other parts of the system may outgrow the comfort zone of programmers and need to be subdivided. The architecture becomes asymmetrical and pragmatic.

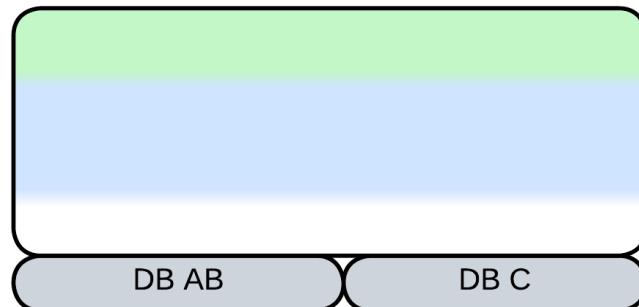
Old age (support) – back to Layers



Back to Layers

When active development ceases, you lose even more people and funding as you drift into the support phase. You are unlikely to retain your best programmers – you'll get novices or even an outsourced team instead. They will struggle to retain the structure of the system – with its mass of hacks from the previous years – against progressively more weird requests from the business and customers whose natural desires have already been satisfied. That will cause many more hacks to be added – and components coupled or merged for the hacks to land – bringing the architecture back to [Layers](#), though this time heavily oversized *layers*.

Death (the ultimate release) – Monolith



Down to Monolith

If the project is allowed to die, it may still have a chance of a final release which aims at improving performance and leaving a golden standard for the generations of users to come. Heavy optimizations will likely require merging the layers to avoid all kinds of communication overhead, reverting the system back to [Monolith](#).

So it goes

Even though I have observed the cycle of architecture expanding and collapsing in embedded software, I believe that these forces apply to most kinds of systems. First you need to go quickly and interfaces are a burden. Then you need the extra flexibility that they provide to reserve space for future design changes. And as the flow of changes ceases, you may optimize the flexibility away to make programming easier and the code smaller and faster. However, the last transition is not always applicable: a distributed system will oppose compacting if it was written in diverse programming languages or needs specialized hardware setups for proper operation.

Going back in time

It can happen that you need to step back through the life cycle – for example, when the domain itself changes drastically: a new standard emerges or the management decides that your application for washing machines fits coffee machines pretty well, as they are basically doing the same things: heating water, adding powder, and stirring – but you have never wrote software for coffee machines before, thus you are back to the R&D phase.

In such cases it may be easier to rewrite the affected components from scratch rather than try to rejuvenate and refit the old code. Remember that you keep your experience – what was originally implemented as an improvised hack will be accounted for in the redesigned architecture. This means that every time a component is rewritten adds to its longevity as its architecture fits the domain more closely and needs fewer hacks (which are inflexible and confusing by definition) to get to production.

Real-world inspirations for architectural patterns

As architectural patterns are usually technology-independent, they must mostly be shaped by the foundational principles of software engineering. And because the same principles are likely at work at every level of a software system, we may expect similar structures to appear on many levels of software, given similar circumstances – which is not always attainable, for the system-wide scope (which means that there are multiple clients and libraries) and distributed nature (which deals with faults of individual components) of many patterns of system architecture don't have direct counterparts in smaller single-process software. Thus we expect to observe a fractal nature for the more generic patterns while narrowly specialized ones are present at only one or two scopes of software design.

Another thought to consider is that it's not in human nature to invent something new – we are much more adept in imitating and combining whatever we see around us. That is why it's so hard to find a genuine xenopsychology in literature or movies – to the extent that the eponymous Alien is just an overgrown [parasitoid wasp](#). Hence there is another pathway to pursue – identifying the patterns which we know from software engineering in the world around us, as the authors of [\[POSA2\]](#) did decades ago.

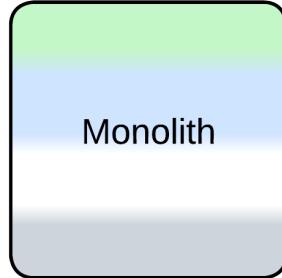
Let's go!

Basic metapatterns

The basic patterns lay the foundation for any system by paving ways to *divide* it into components to *conquer* its [complexity](#). We are going to observe them all around:

Monolith

A *Monolith* is a (sub)system the internals of which we prefer not to see



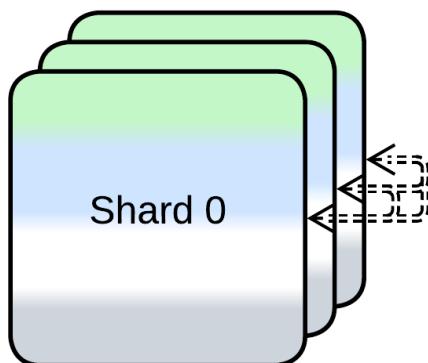
It may lack an internal structure or its components may be coupled

[Monolith](#) means encapsulation – we use the thing without looking inside:

- You interact with your dog (or your smartphone) through their interface without thinking of their internals.
- A function exposes its name, arguments and, probably, some comments. The implementation is hidden from its users.
- An object has a list of public methods.
- A module or a library exports several functions for use by its clients.
- A program is configured through its command line parameters and managed through its [CLI](#). We don't care how the Linux utilities (e.g. *top* or *cat*) work – we just run them.
- A whole distributed system may be hidden behind a web page in your browser – and you never imagine its complexity unless you have worked on something of a kind.

Shards

Shards are multiple instances of a subsystem



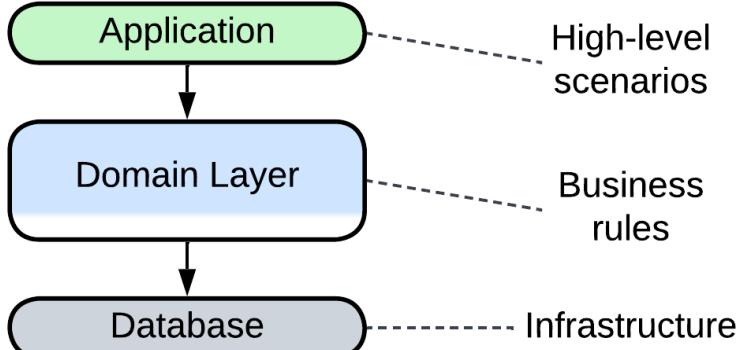
A shard may be stateless or stateful

Shards is about having multiple instances of something, which often differ in their data:

- A company employs many programmers to accelerate development of its projects.
- Carrying two mobile phones from different operators fits this pattern as well.
- This is how they make modern processors more powerful: by adding more cores, not by running them faster.
- Objects in OOP are the perfect example of having multiple instances that vary in their data.
- Running several shells in Linux is a kind of sharding.
- A client application of a multi-user online game is a shard.

Layers

Layers divide the system by the level of abstractness

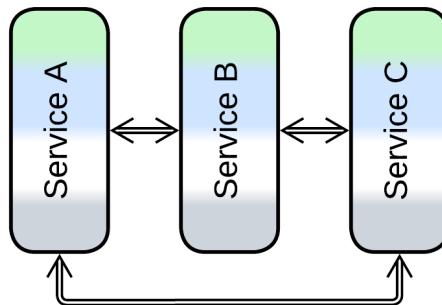


Layers is the separation of responsibilities between external and internal components:

- In winter we wear soft clothes on our body, a warm sweater over them, and a wind-proof jacket as the external layer.
- An object comprises high-level public methods, low-level privates, and data.
- An OS has a UI which runs over user-space software over an OS kernel over device drivers over the hardware.
- Your web browser executes a frontend which communicates to a backend which uses a database.

Services

Each service implements a subdomain



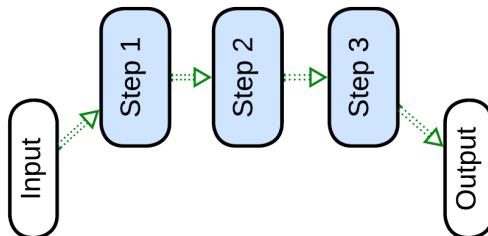
The subdomains should be loosely coupled

[Services](#) boil down to composition and [separation of concerns](#):

- We have legs, arms, and other specialized members.
- A gadget contains specialized chips for activities it supports.
- [GoF] advocates for an object to incorporate smaller objects (composition over inheritance).
- Applications often delegate parts of their logic to specialized modules or libraries.
- An OS dedicates a driver for each piece of hardware installed. Moreover, it provides many tools to its users – instead of tackling all the user needs within the kernel.
- [DDD] describes the way to subdivide a large system into loosely coupled components.

Pipeline

The system processes data in a sequence of steps



Each step knows nothing of its neighbors

[Pipeline](#) is about the stepwise transformation of data:

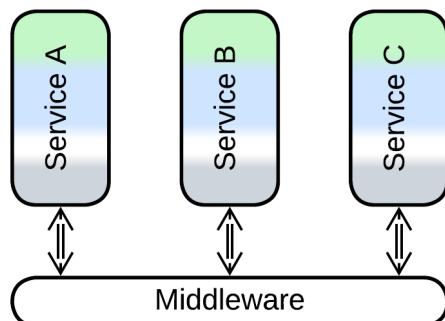
- The pattern got its name from real-world plumbing.
- You'll see similar arrangements in [cellular metabolism](#).
- It is the basis for [functional programming](#).
- Linux command line tools are often skillfully composed into pipelines.
- Hardware is full of pipelines: from [CPU](#) and [GPU](#) to audio and video processing.
- Finally, a UI wizard passes its users through a series of screens.

Extension metapatterns

An extension pattern encapsulates one or two aspects of the system's implementation. It may appear only on design levels which have those particular aspects:

Middleware

The *Middleware* provides communication for the services



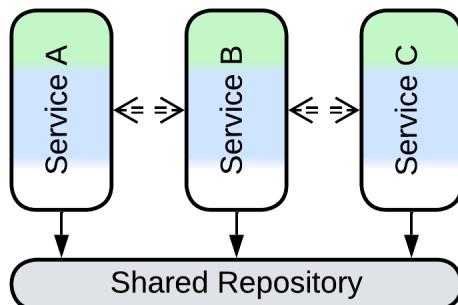
It may also manage their instances

A *Middleware* abstracts scaling and/or intercommunication:

- The network of post offices is a middleware – you push the letter into a mailbox and it automagically appears at its destination's door.
- A *bus depot* may mean a bus garage which deploys as many buses as needed to service the traffic or a bus station where people come to have a ride, regardless of the exact vehicle model they'll take.
- Hardware is full of another kind of *buses* that unify means of communication.
- TCP and UDP sockets hide the details of the underlying network.
- A distributed *actor framework* allows an actor to address another actor without knowing where it is deployed.

Shared Repository

The *Shared Repository* owns the system's data



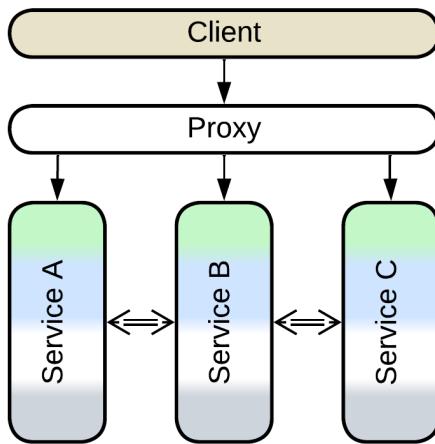
The services communicate directly or through the repository

A *Shared Repository* provides data storage and/or data change notifications:

- Everybody in the room may use a blackwhiteboard to express and exchange their ideas.
- An Internet forum works in a similar way – people post their arguments there for others to see them and get notified on answers.
- RAM and CPU caches are kinds of shared repositories. CPU caches are [kept synchronized through notifications](#).
- *Observer* [GoF] is about getting notified when a shared object changes.
- Services or service instances may share a database.

Proxy

A *Proxy* stands between a (sub)system and its clients



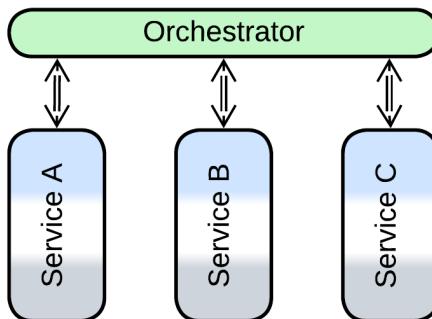
It implements a generic aspect of the system's behavior

A *Proxy* isolates a system from its environment by translating between internal and external protocols and/or implementing generic aspects of communication:

- You may need a translator to understand foreign people or have a secretary to deal with routine tasks. A local guide combines both roles.
- An adapter makes several hardware plugs (or software frameworks) compatible.
- Your Wi-Fi router is a proxy between your laptop and the Internet.
- A compiler is a kind of proxy between source code and bytecode.

Orchestrator

The *Orchestrator* runs high-level scenarios by using the services



It also keeps the data of the services consistent

An *Orchestrator* integrates several components by implementing high-level use cases and/or keeping them in sync:

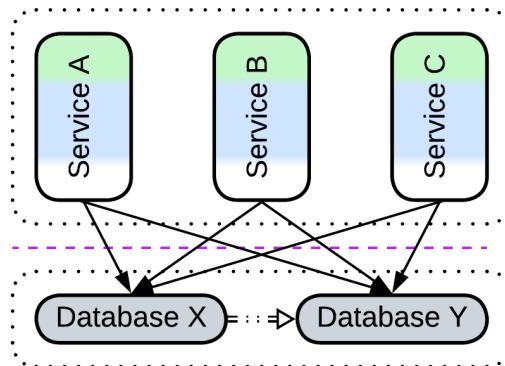
- A taxi driver orchestrates his car's internals.
- A *Facade* [GoF] provides a high-level interface for a system while a *Mediator* [GoF] integrates a system by spreading changes initiated by the system's components.
- A linker composes a working program out of disjunct modules.

Fragmented metapatterns

A fragmented pattern uses small specialized components to approach a case which is hard to resolve with more generic means. The high degree of specialization leads to even fewer examples:

Polyglot Persistence

The system gains benefits from multiple database technologies



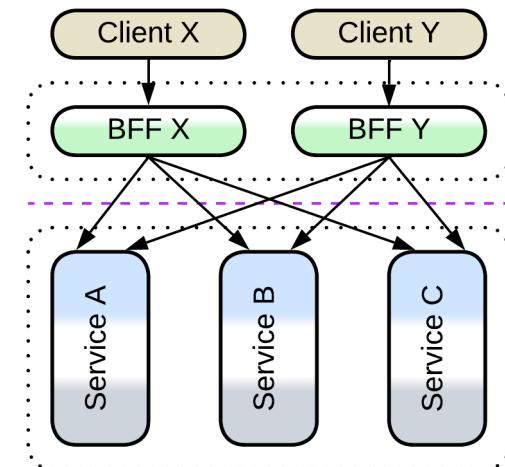
The databases store separate subsets or derived views of the data

[Polyglot Persistence](#) is about having multiple containers for data:

- A warehouse or a cargo ship has dedicated storage areas with extra facilities for combustible, toxic, and frozen goods.
- A computer has CPU caches, RAM, flash, and hard drives for temporary or permanent data storage.
- There are map, list, and array – each with its pros and cons. A large class would often use two or three kinds of containers and not without reason.

Backends for Frontends

Each BFF is dedicated to its kind of client



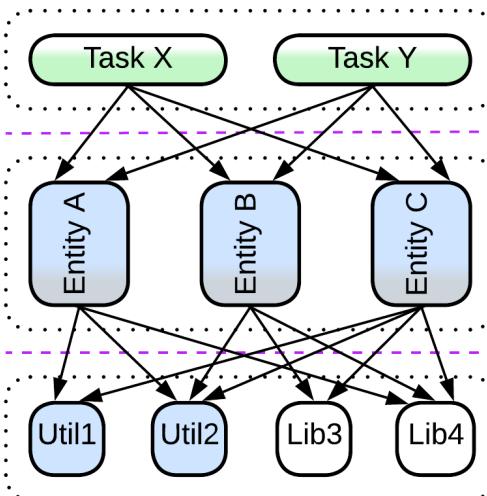
Each BFF orchestrates all the services

[Backends for Frontends](#) is about treating different kinds of clients individually:

- A bank is likely to reserve a couple of employees to serve rich clients.
- A Wi-Fi router has many management interfaces: web, mobile application, CLI, and probably [TR-069](#).
- A multiplayer game may provide desktop and mobile client applications.

Service-Oriented Architecture

SOA is 3 or 4 layers of services



Each task orchestrates entities

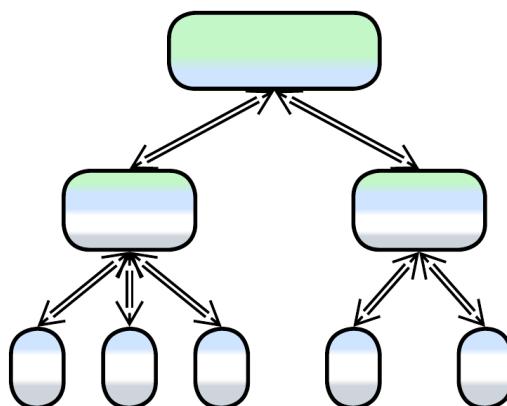
Entities use libraries and utilities

[SOA](#) applies OOP techniques, including component reuse, to deal with complex systems:

- That's what you have inside your car. Many of its internals rely on the car's battery for power supply instead of having a small battery installed inside every component.
- Cities are built in the same way – schools, markets, and railways serve multiple houses.
- It's the same with user space of operating systems: there is a shared UI framework which interfaces with as-many-as-needed applications, each of which calls shared libraries (DLLs).

Hierarchy

Hierarchy is a tree of components



Each root orchestrates its children

Child nodes may be specialized or polymorphic

[Hierarchy](#) distributes system's complexity over multiple levels:

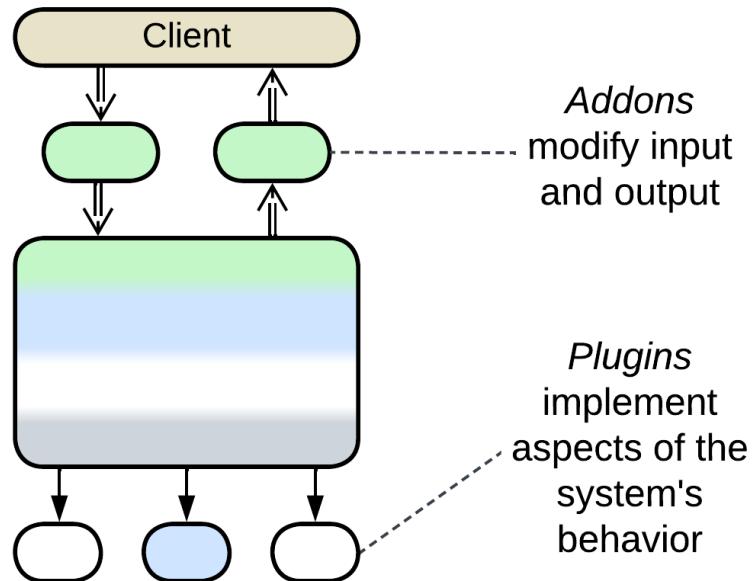
- This is how large companies and armies are managed.
- Large projects are made of services which contain modules which contain classes which contain methods.

Implementation metapatterns

An implementation pattern highlights the peculiar internal arrangements of a component. Such patterns are deeply specialized:

Plugins

Plugins allow for customization of the system's functionality

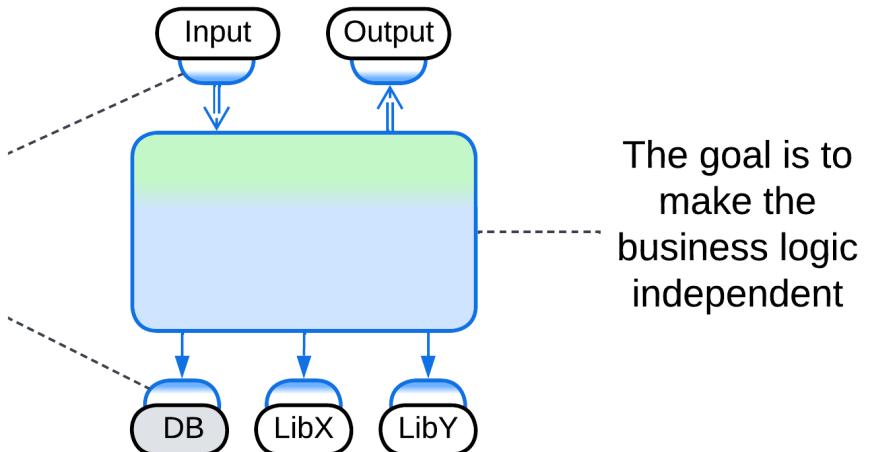


Plugins make a component's behavior flexible through delegating its parts to small external additions:

- This is how we use tools for our work – a man becomes a digger when given a shovel.
- *Strategy* [[GoF](#)] is the thing.

Hexagonal Architecture

Hexagonal Architecture assigns an adapter to each external component

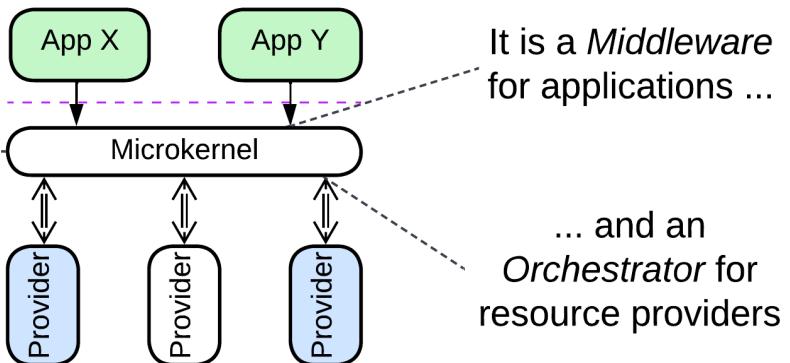


Hexagonal Architecture protects the internals of a system from its environment:

- A drill or a screwdriver has replaceable bits.
- *OS Abstraction Layer* and *Hardware Abstraction Layer* in embedded systems or *Anti-Corruption Layer* in [[DDD](#)] are all about that.

Microkernel

The *microkernel* shares resources of providers among applications

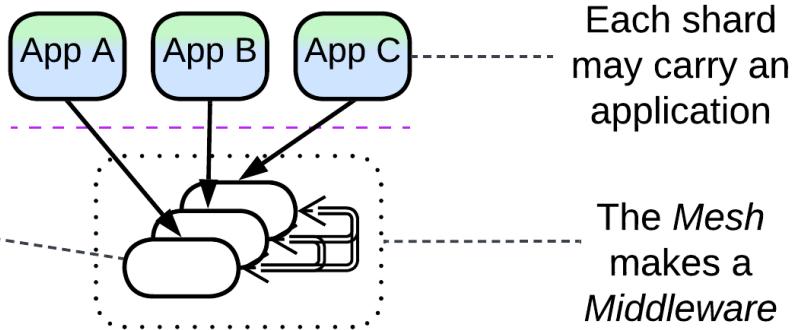


Microkernel shares the goods of resource providers among resource users:

- It's like a bank that takes money from the rich to distribute them among the poor.
- This is what an OS is for. Its scheduler shares the CPU, the memory subsystem shares RAM, while the device drivers provide access to the peripherals.
- Cloud services are based on sharing computation resources among clients.

Mesh

Mesh comprises interconnected Shards



Mesh is like grassroots movements – self-organizing and survival through redundancy:

- Ants and bees are small, autonomous, and efficient. Their strength comes from their numbers.
- Road networks and power grids don't collapse if some of their components are damaged as they are highly redundant.
- Torrents, mobile communications, and the Internet infrastructure are known for their robustness.

Summary

Architectural patterns have parallels in the natural world, our society and/or different levels of computer hardware and software. Learning about them helps us feel the driving forces behind patterns and be more flexible and creative in using the patterns we know and devising new ones.

The heart of software architecture

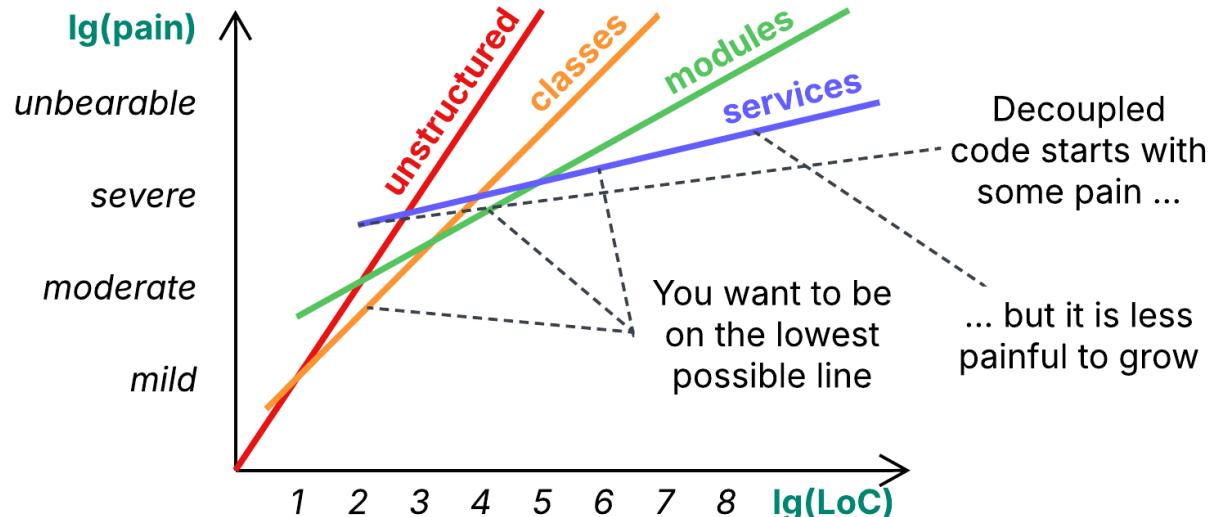
As the visible world boils down to protons and neutrons which stick together in various combinations, so too does the entirety of software architecture as a field of endeavor grow from an interplay of *cohesion* and *decoupling*.

Cohesers and decouplers

Any project carries many constraints (*forces*), some of which want for certain parts of its code to be kept together (*cohesive*) while others push to have them torn apart (*decoupled*). Their balance and the resulting optimal architecture is very fluid as each of the forces in action depends on the current circumstances, the project's history and its expected evolution.

Code structure and the level of pain

Let's explore how a force influences the structure of a project. Consider the *clarity of code* which determines *development velocity*:



When you have 10 lines of business logic, you are likely to write them down as a simple script. Separating them into classes or deploying 5 services, each running 2 lines of code, is an overkill which would make the complexity of your infrastructure much higher than that of the task on hand.

At 100 lines of code you are likely to be more comfortable with procedures or even classes to divide the code into, as keeping everything together starts to hurt. You switch from the most cohesive implementation to another one, decoupled to an extent. Though the latter is more complex at its core, it allows for less painful growth because it encompasses smaller components.

A file of 5 000 lines is hard to read – you need to separate it into modules, each of which contains classes, which contain methods, which contain the code. You are building yet another level of the hierarchy to keep the number of items in each piece (lines in a method, methods in a class, classes in a module) comfortably small.

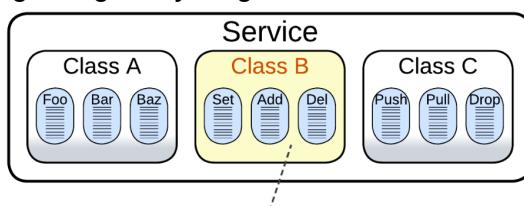
At about 100 000 lines you may start considering further separation of your project into services – as, you know, there are merge conflicts, or it takes a while to compile and test the

whole codebase... anyway, at that point very few people comprehend the whole thing in detail, thus the benefits of having the entire codebase co-located ([monorepo](#)) are diminishing while new drawbacks emerge.

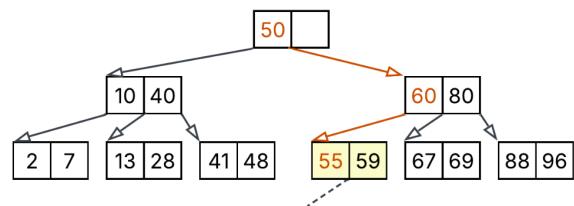
Building a hierarchy

As we see from the example above, code clarity favors *cohesiveness* (everything together) for smaller codebases but *decoupling* (parts separated with narrow interfaces) for larger projects. We can state that the direction the force under review (clarity) pushes us in depends on the project's size. That is not a unique case – many forces work this way, resulting in the famous [Monolith vs Microservices complexity diagram](#).

Such a behavior is common for forces and, by the way, it is also the case with editing sorted data. Array is the most efficient data structure for a small collection (up to about 1 000 elements) while anything larger requires a hash map or [B-tree](#) (hierarchy of arrays). Just as a database splits oversized arrays because they are too slow to edit, the human mind is inefficient with large collections of similar items and wants them to be restructured into a hierarchy. When we look into a service, we see only the classes it contains. When we examine a class, we check the list of its methods, not those of surrounding classes. And when we open a method, we try to understand how its lines of code work together. This is the way humans fight complexity – by selecting a segment at one level of abstraction and ignoring everything around.



You are reading this



The database is editing this

A hierarchy inherently adds some inconvenience – traversing levels of a B-tree slows down operations, and a project with many files takes time to grasp – which is why we avoid deep hierarchies in smaller projects (or datasets) – but that is still a very low cost for having any individual component (a method, class or module in a project; an array in a B-tree) stay reasonably small and simple thanks to the distribution of the overall complexity (or data) over the hierarchy.

Bidirectional forces

Among the forces that prefer decomposing a project into segments of certain size are:

- *Clarity* – as discussed above.
- *Development velocity* – programmers are [very productive](#) when they know their code and don't waste their time on communication. Still, there is a productivity limit for a single person, and if you want your project to be developed faster than that, you need to divide it among several teams, sacrificing individual performance for brute force.
- *Latency* – it is poor for a distributed system, but is not much better for a large monolith that may deadlock. Ideally, you should separate the latency-critical part into a dedicated small component placed close to the input and output hardware.
- *Throughput* – on one hand, interservice communication is suboptimal because of the associated networking and serialization. On the other hand, there is a limit for a single server's performance as more powerful hardware is too expensive. It is

cheaper to install several commodity servers and accept a communication penalty than keep the entire system running on a single high-end machine to avoid networking.

- *Security* – a single process is easier to audit and secure than a system of services. However, as a service grows, it may develop multiple interfaces and assimilate many libraries. Eventually, protecting that mishmash becomes much harder than creating a secure perimeter around a distributed system.

Cohesers

Other forces predominantly push you towards merging all your code and data together:

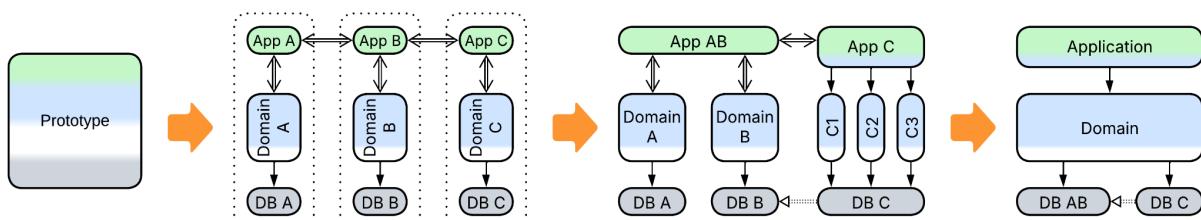
- *Debuggability* – it is hard to debug a distributed system. You need to investigate logs and attach your debugger to several services which may be written in different programming languages. Debugging a single process, even 10 MLoC in size, is almost always easier (except when it is undocumented).
- *Data consistency* – when everything runs in a single thread, you don't have to care about data races, lost packets, or [idempotence](#). The [CAP theorem](#) is on your side.
- *Data analysis* – it is hard to collect data from multiple sources. You cannot use SQL joins. However, at some point in the distant future, you may still reach the performance limit of your single database, in theory making data analysis a kind of bidirectional force.

Decouplers

And there are forces that try to keep your code fragmented:

- *Variability* – if your project needs to satisfy many [conflicting requirements](#), it is very hard to achieve that with a uniform codebase running in a single process.
- *Location* – you may need to run parts of your system on [its end users' devices](#), in [regional data centers](#), or even on [specialized hardware](#).
- *Organizational structure* – according to [Conway's law](#), forcing everyone to work on a shared component is among the top team performance killers, especially when time zone differences are involved.

Expansion and contraction



As was [discussed previously](#), when you start a project by building a [PoC](#) or prototype, you have little code and need to move quickly. Most of the *decouplers* are not there and the *bidirectional forces* favor cohesion, thus you don't waste your time on extra interfaces or fine-grained services. You don't have multiple teams to fall prey to Conway's law.

As soon as the prototype is approved, it's time to prepare for iterative development of a project which nobody comprehends in advance. Requirements will change many a time. Libraries and frameworks may not work as expected. External services may die or change. Your team members may leave or, worse, be eager to try a newer technology. You need all

the flexibility in the world, and the flexibility comes through decoupling. But you also rely on your speed to remain ahead of competitors, therefore you cannot go too far because decoupling is not free. You try to imagine what may change in the future and prepare by isolating the would-be affected parts with well-defined interfaces.

As your software matures, the flow of changes slows down and the business becomes more predictable or, rather, more familiar. You observe that some of the interfaces which you've added have never been put to real use – they just make the project more complex – you pay for decoupling without earning its benefits. Others have been found to be too restrictive and were removed. Thus the system begins to contract, burning the flexibility it no longer needs while also growing in directions that were not originally expected.

Finally, you move to the support phase. The best programmers leave for more active projects. Whoever replaces them does not know the code. The remaining flexibility decomposes in favor of quick hackarounds. What remains is an ugly cohesive [evolutionary-shaped mess](#), created through generations of ad-hoc patches.

Deconstructing patterns

Imagine a dungeon with dragons. It is made of halls connected by tunnels. Each hall is *cohesive*. Tunnels are narrow interfaces that *decouple* them. A hall is amorphous – it can have any shape but it cannot open to another hall except through a tunnel – such are the rules of the game. The tunnels both restrict the freedom of the halls and interconnect them.

SOLID principles

If *cohesion* and *decoupling* dictate software architecture, they should surface in its principles. Let's take a look at [SOLID](#):

- The *single responsibility principle*, also known as [do one thing and do it well](#), is a general advice for keeping unrelated functionality decoupled.
- The *open-closed principle* and *Liskov substitution principle* decouple the logic of the parent class or the code that uses it, correspondingly, from the functionality of its subclasses.
- The *interface segregation principle* decouples independent parts of an object's interface.
- The *dependency inversion principle* decouples an object's users from its implementation.

Please beware that each of those principles in and of themselves involves decoupling which is not free – your software may end up having too many moving parts and strict rules to remain easy to read and support.

When we choose between cohesion and decoupling, we choose between a single component and a pair of components connected through a constraint rule. The more decoupling, the more components and rules we have to handle. Sooner, rather than later, the number of individual components and rules will overwhelm any developer.

Gang of Four patterns

Let's now discuss something more practical, namely the [\[GoF\] patterns](#) which seem to be ingenious but hacky ways for rearranging the roles in your code. They override ordinary

OOP rules, which is useful when you need extra flexibility. For example, the *creational patterns* interfere with the normally cohesive *select type – create – initialize – use* sequence of operating an object.

Some patterns provide a basic decoupling:

- *Adapter* translates between two interacting components so that they may evolve independently.
- *Observer* decouples an event from the reactions it causes by registering handlers at runtime.
- *Chain of Responsibility* separates method invocation from method execution. A client's calling a method of an object runs the corresponding method of another object.

Others break the functionality or data of a class into two or more parts, juggling them at runtime:

- *Proxy* separates an object's representation from its implementation, enabling lazy loading or remote access.
- *Flyweight* segregates an immutable data member of a class to save memory by merging multiple instances of identical data.
- *Strategy* and *Decorator* decouple a dimension of an object's functionality to allow runtime changes in or composition of the object's behavior, respectively.
- *State* separates an object's behavior into multiple classes based on the object's state.
- *Template Method* decouples several aspects of a class's behavior from its main algorithm and envelops variations of those aspects into subclasses.
- *Bridge* separates a high-level hierarchy of classes from their low-level implementation details which may comprise an orthogonal hierarchy.
- *Memento* decouples the lifetime of an object's state from the object itself.

On the other hand, a few patterns gather separate components together:

- *Command* collects all the data required to call a method.
- *Mediator* is a cohesive implementation of multi-object use cases.
- *Composite* and *Facade* represent multiple objects as a cohesive entity. A *Composite* broadcasts a call to its interface to every object it contains while a *Facade* orchestrates the wrapped subsystem.
- *Abstract Factory* and *Builder* encapsulate *type selection* and *initialization* for several related hierarchies, so that the client code gets objects from a set of consistent types. On top of that, a *Builder* cross-links the objects it creates into a cohesive subsystem, which is returned to the *builder*'s client as a whole.

The remaining patterns pick an aspect or two of an object's behavior and move them elsewhere:

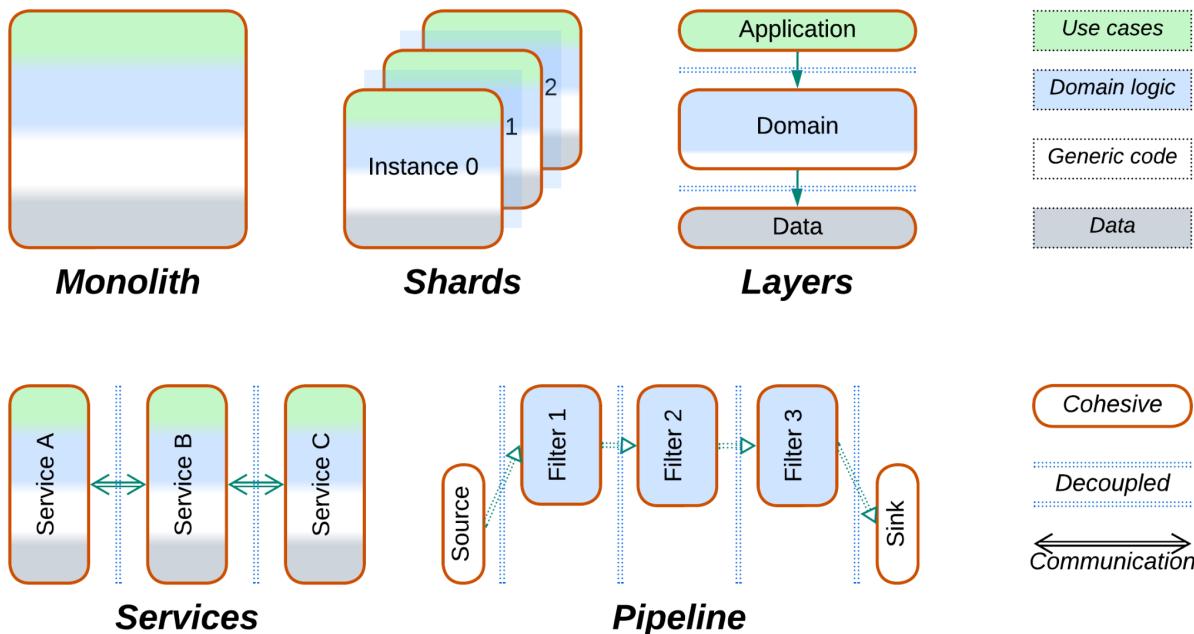
- *Iterator* moves the code for traversal of a container's elements from the container's clients into the container's implementation, decoupling its clients from the iteration algorithm.
- *Visitor* collects actions that a client needs to perform on each kind of object in a hierarchy, decoupling them from the classes that constitute the hierarchy.

- *Interpreter* decouples client scenarios from the rest of the system by having them written in a dedicated language and run in a protected environment.
- *Prototype* binds the *type selection* and *initialization* together and decouples them from the object *creation*.
- *Singleton* binds the *creation* and *initialization* of a global object to every call of its methods.
- *Factory Method* decouples the *initialization* from *type selection* and hides both from the class's users.

As we see, every [GoF] pattern boils down to binding (making *cohesive*) and/or separating (*decoupling*) some kind of functionality or responsibilities.

Architectural metapatterns

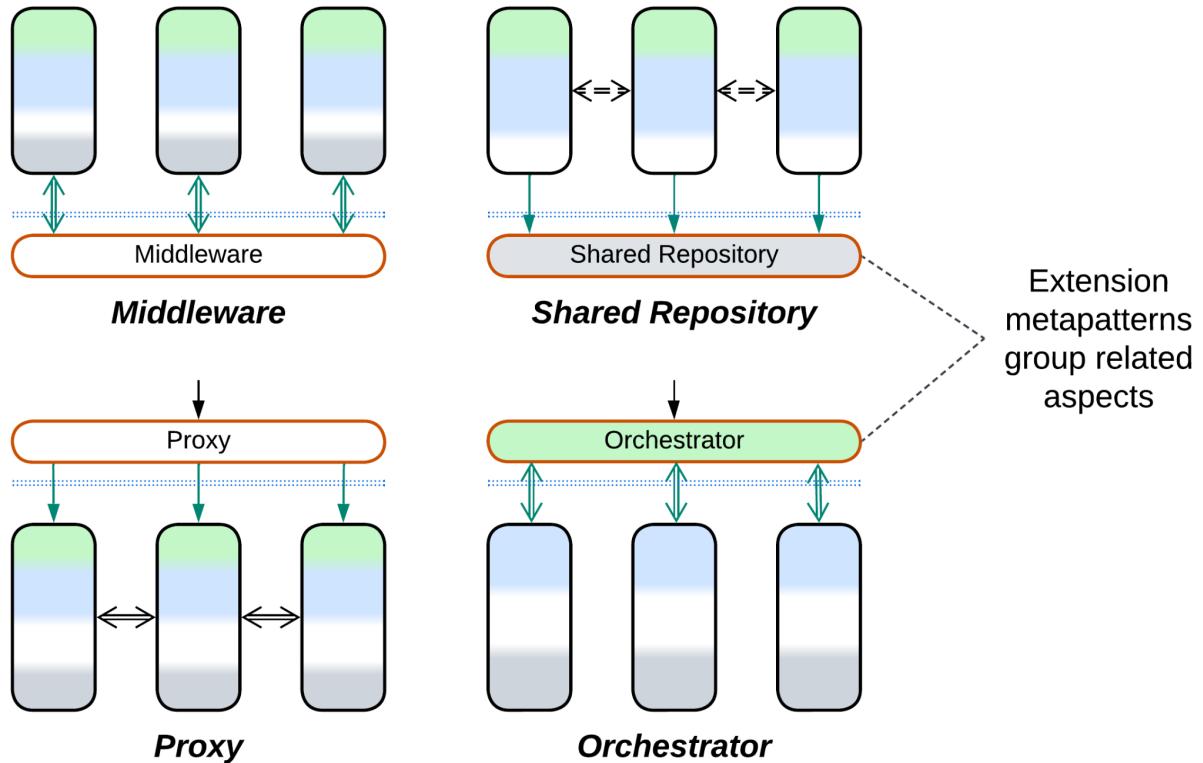
Finally, let's close the book by iterating over the metapatterns and looking into their roots through the lens of unification and separation.



Basic architectures:

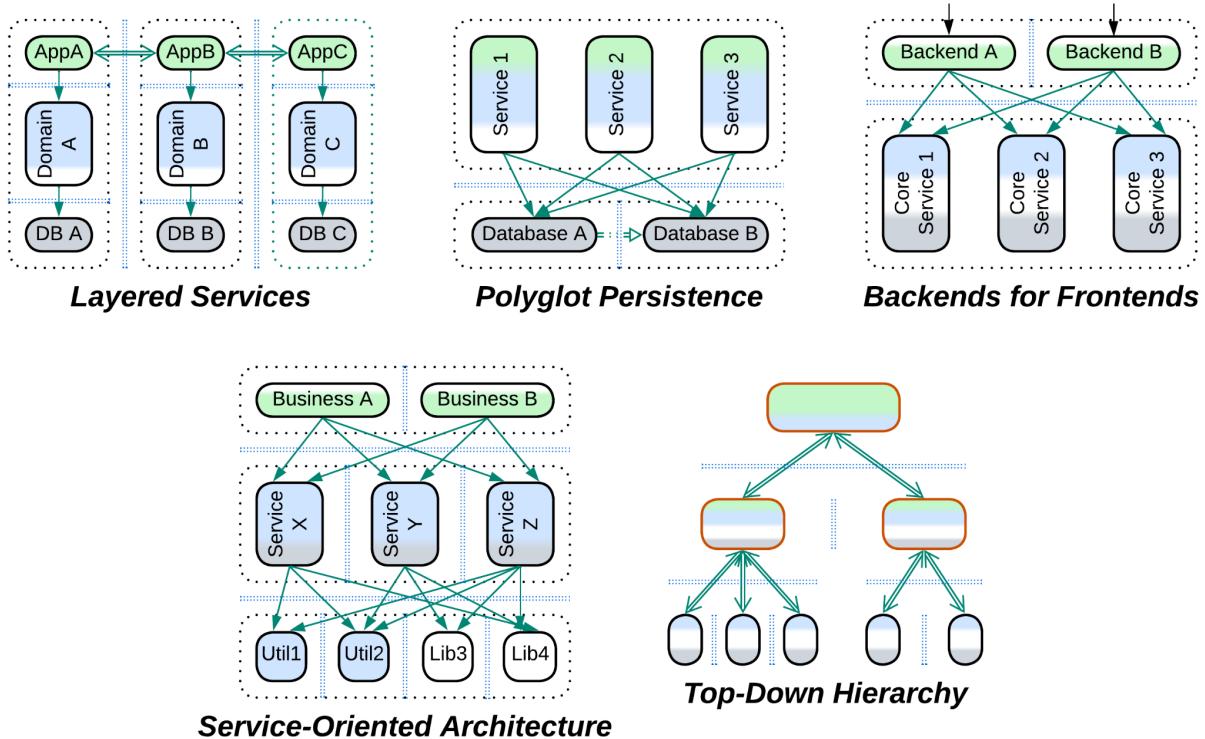
- Monolith keeps everything together for quick and dirty projects:
 - Total cohesiveness results in low latency, cost-efficient performance, and easy debugging.
- Shards slice a large-scale application into multiple instances:
 - Decoupling the instances enables scaling but sacrifices consistency of shared data.
- Layers separate the high-level code from low-level implementation:
 - Cohesion within a layer makes it easy to implement and debug.
 - Decoupled layers may vary in technologies and properties but are somewhat slower and hard to debug in-depth.
- Services divide a complex system into subdomains:
 - Cohesiveness of a service keeps it simple and efficient when it does not need to consult with other services.

- *Decoupling* enables development of larger codebases by multiple specialized teams but global use cases become complicated.
- [Pipeline](#) segregates data processing into self-contained steps:
 - *Decoupling* simplifies reassembling or expanding the system but increases its latency.



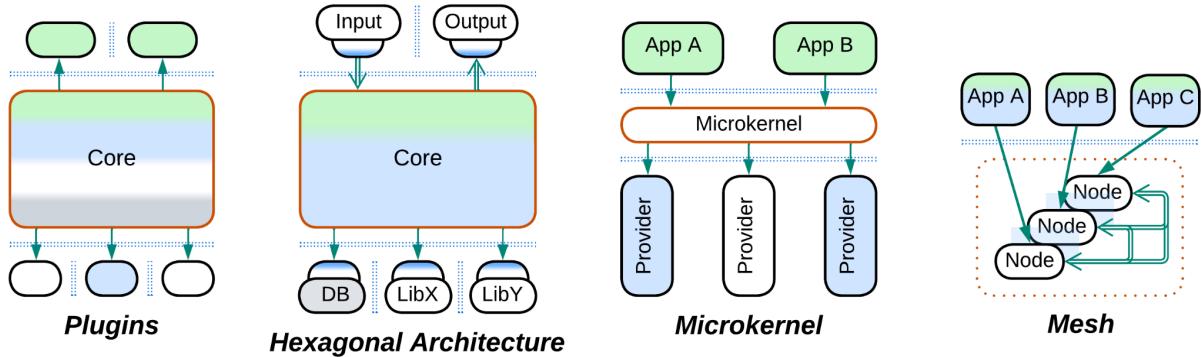
Grouping related functionality:

- [Middleware](#) separates the implementation of communication and/or instance management from the business logic:
 - The *cohesive* communication layer is reliable and uniform, thus it is easy to learn.
 - *Decoupling* communication concerns from the business logic simplifies the latter.
- [Shared Repository](#) dissociates data from code, enabling [data-centric programming](#):
 - Cohesive data is consistent and easy to handle.
 - *Decoupled* business logic can be scaled or subdivided [independently](#) of the data.
- [Proxy](#) mediates between a system and its clients, taking care of one or more aspects of their communication:
 - A *cohesive* edge component is easier to manage and secure.
 - *Decoupling* generic aspects simplifies business logic but usually increases latency.
- [Orchestrator](#) collects a multitude of complex use cases into a dedicated layer:
 - Cohesive use cases are easy to comprehend and debug.
 - *Decoupling* use cases from domain logic allows for variation in technologies but increases latency and complicates in-depth debugging.
- [Combined Component](#) blends two or three of the above layers:
 - *Cohesion* improves performance but reduces flexibility.



Decoupled systems:

- [Layered Services](#) first decouple the subdomains, and then the layers within each subdomain:
 - *Decoupled* subdomains allow for multi-team development and large codebases but complicate global use cases. *Decoupled* layers enable variation in technologies within a subdomain and [limit interdependencies](#) between subdomains to a single layer.
- [Polyglot Persistence](#) divides data among multiple data stores:
 - *Decoupling* improves performance through database specialization at the cost of consistency.
- [Backends for Frontends](#) dedicate one or two components (a [Proxy](#) and/or [Orchestrator](#)) per each kind of client.
 - *Decoupling* allows for customization on a per-client-type basis but makes it hard to share functionality among the clients.
- [Service-Oriented Architecture](#) first segregates a large system into layers, then subdivides each layer into services:
 - *Decoupling* layers strangely enables reuse as any component of an upper layer can access every component below it. *Decoupling* services within the layers allows for multi-team development. Drawbacks include high latency, system complexity, and interdependencies.
- [Hierarchy](#) recursively separates general and specialized logic, tackling complexity:
 - *Cohesive* general and subdomain-specific business logic helps readability and debugging.
 - *Decoupled* layers and subdomains allow for modification and expansion of local functionality at the cost of performance.



Component implementation:

- [Plugins](#) separate customizable aspects of a system's behavior:
 - *Decoupling* several aspects of a system allows for it to be fine-tuned but requires careful design and may lower performance.
- [Hexagonal Architecture](#) isolates the business logic from its external dependencies:
 - *Decoupling* protects from vendor lock-in and supports automatic testing at the cost of lost optimization opportunities.
- [Microkernel](#) mediates between resource consumers and resource providers:
 - *Cohesive* resource management optimizes resource usage.
 - *Decoupling* allows for seamless replacement of resource providers.
- [Mesh](#) aggregates distributed components into a virtual layer:
 - *Virtual cohesion* hides the complexity of distributed communication from client code.
 - *Actual decoupling* (distribution) of the nodes enables scaling and fault tolerance.

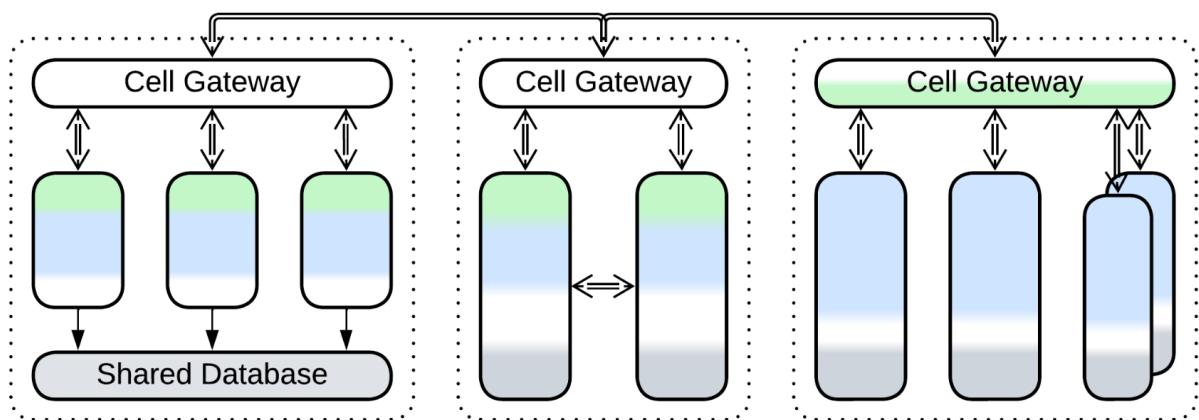
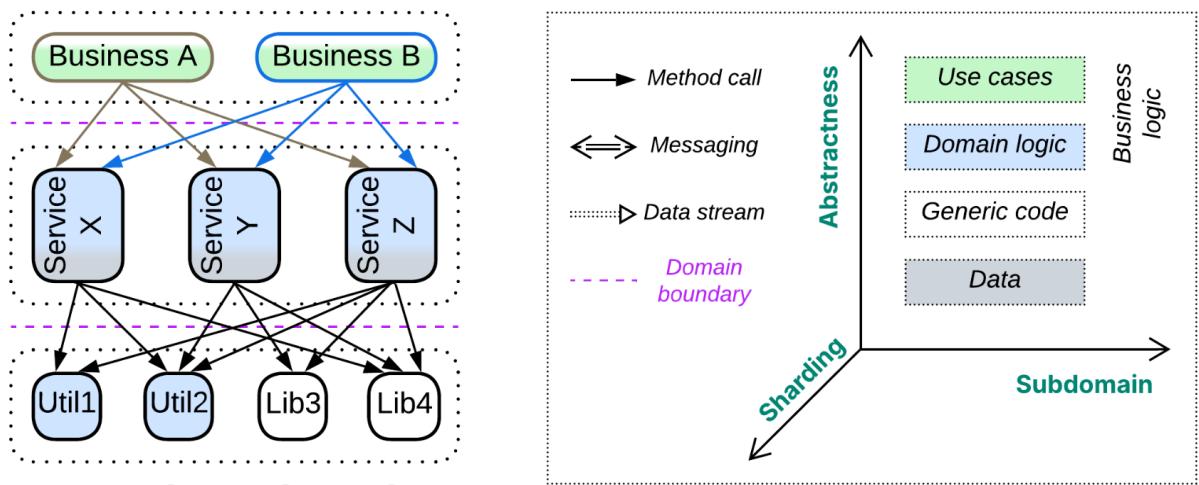
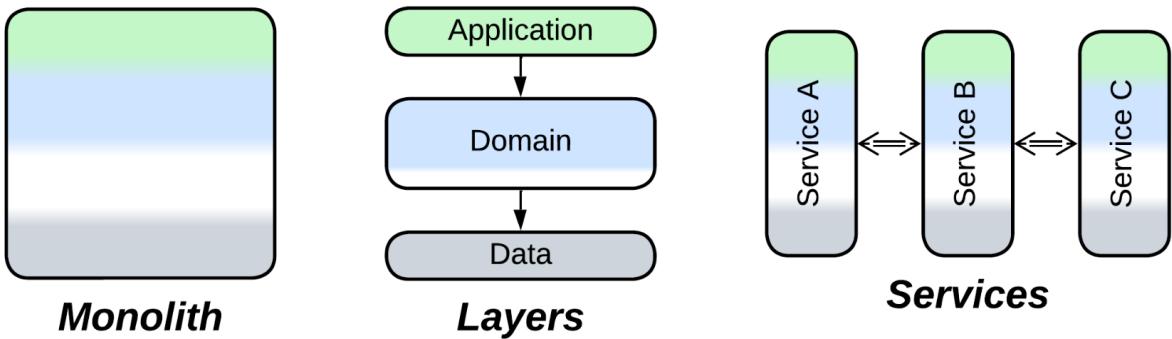
Choose your own architecture

Now that we've seen patterns decomposed into decoupling and cohesion, we can try reconstructing architecture based on your project's needs.

Project size

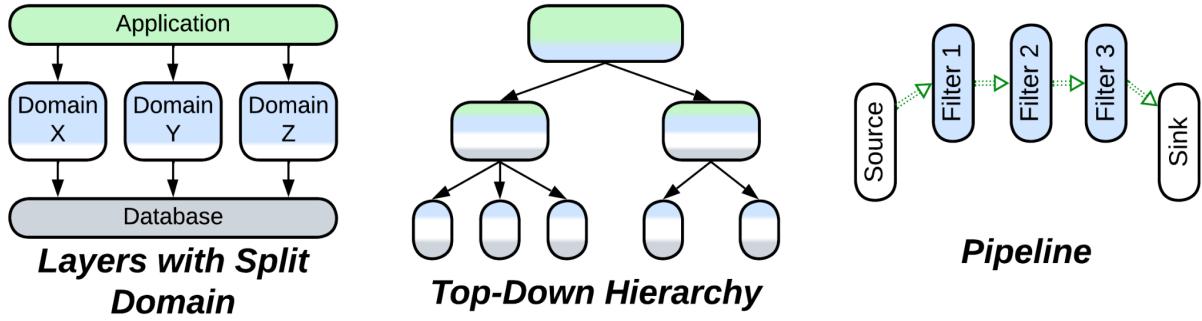
The project's expected size is among the main determinants of the project's architecture as both overgrown components and excessive fragmentation handicap development and maintenance. A moderate number of components of moderate size is the desired zone of comfort.

Therefore, a one day task will likely be [monolithic](#), a man-month of work needs [layering](#) while anything larger calls for at least partial separation into [subdomain modules](#) or [services](#). Very large projects may require further subdivision into [Service-Oriented Architecture](#) (SOA) or [Cell-Based Architecture](#) (a kind of [Hierarchy](#)).



Cell-Based Architecture

Any inherent decoupling within your domain is another factor to consider in the initial design. For example, the layer with domain logic is very likely to contain independent subdomains which naturally make modules or services at next to no development or runtime cost. Likewise, [Top-Down Hierarchy](#) is a good fit for a hierarchical domain. A domain that builds around stepwise processing of data or events may be modeled as a [Pipeline](#), which is a very flexible architectural style.



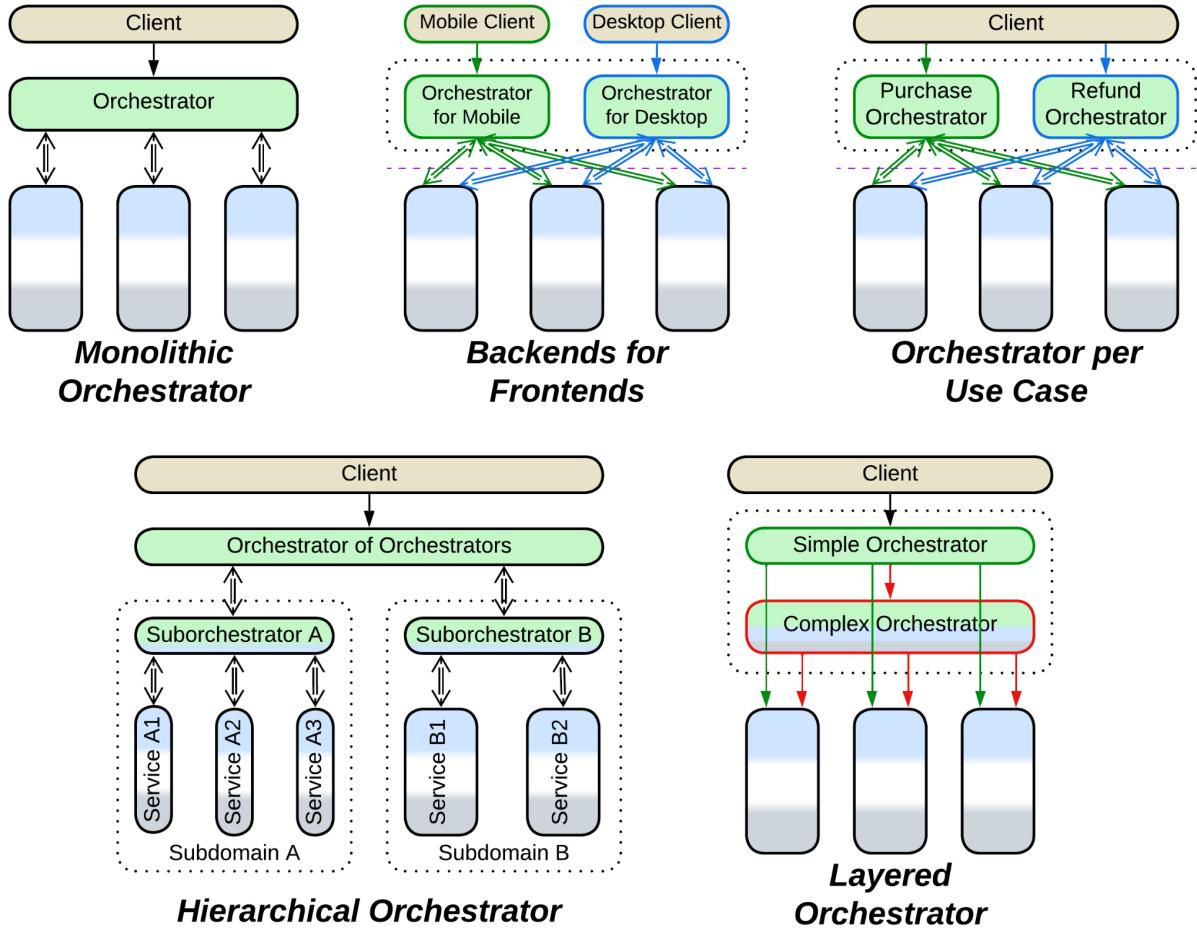
The number of teams you start the project with is also important. For the teams to be as efficient as possible you want them to be almost independent. As every team gets ownership of one or two components, you must assure that the architecture has enough modules or services for the teams to specialize, because anything shared will likely become a bottleneck. For example, you can hardly employ more than 3 teams with a [layered architecture](#) as there are only so many layers in any system. Thus, having a large number of teams strongly hints at [Services](#), [Pipeline](#), [SOA](#), or [Hierarchy](#).

If not all the teams are available from day one, it is still preferable to initially set up component boundaries for the prospective number of teams because subdividing an already implemented component is a terrible experience. However, it may be [easier and safer](#) for now to leave all the components running within a single process (as [modules](#)) to avoid the overhead of going distributed and have less trouble moving pieces of code around as needed (as new requirements often make a joke of your original design). You should be able to make *modules* into [services](#) through moderate effort once that becomes an imperative.

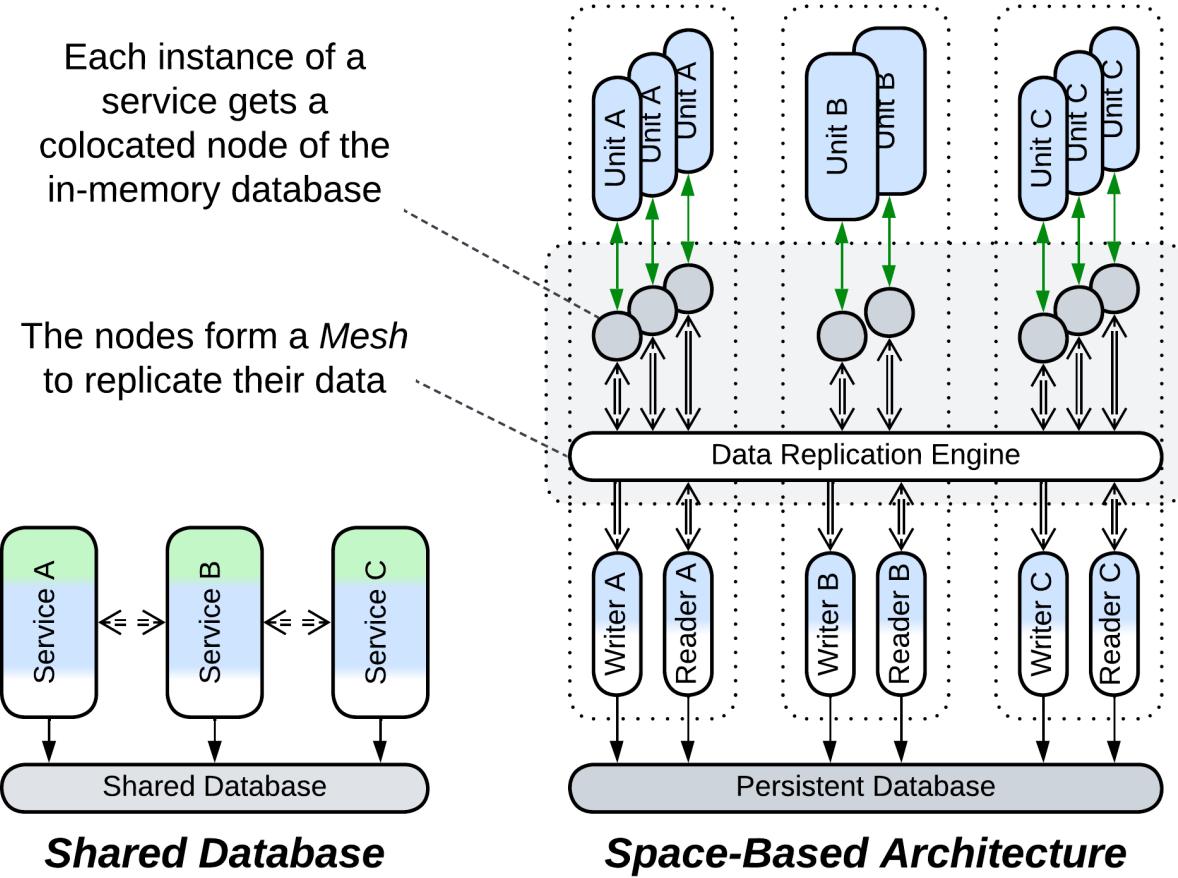
Domain features

We've already seen above that hierarchical or pipelined domains enable the use of corresponding architectures. There is more to it.

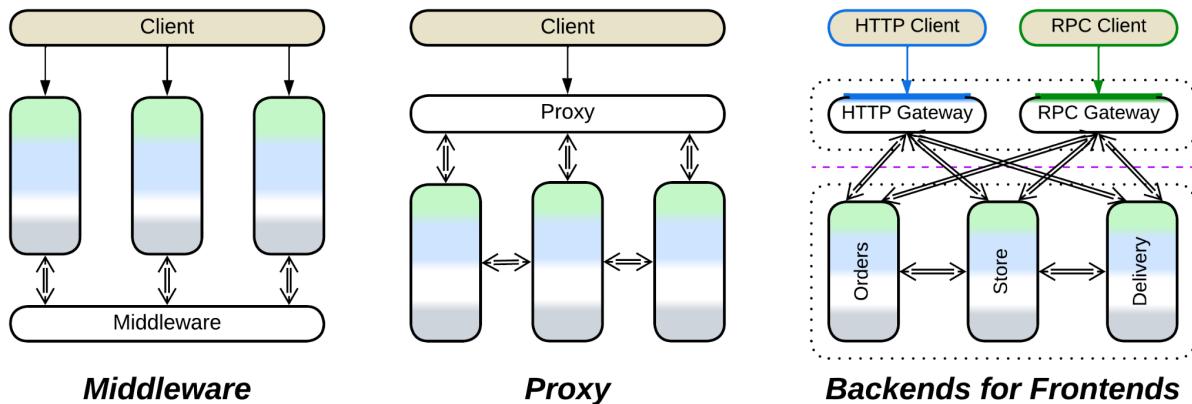
Sometimes you expect to have many complex use cases which cannot be matched to your subdomains because every scenario involves multiple components, thus spreading over the entire system. You would usually collect the global use cases into a dedicated component – an [Orchestrator](#). And if the *Orchestrator* grows out of control, it is [subdivided](#) into layers or services.



Other systems are built around data. You cannot split it into private databases because almost every service needs access to the whole which necessitates a [Shared Repository](#), or the highly performant [Space-Based Architecture](#).



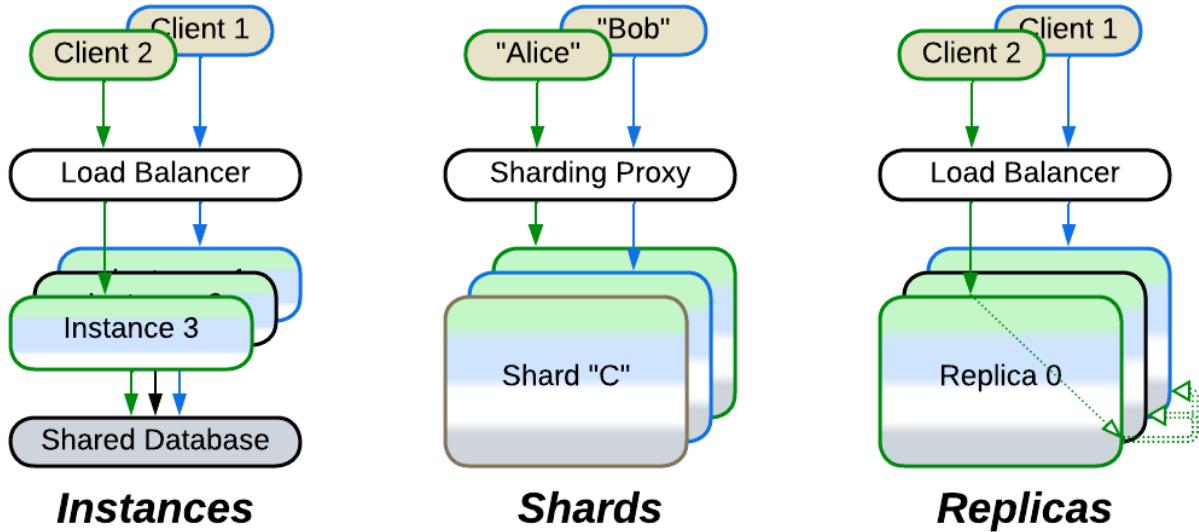
Once you go distributed, you will likely employ a [Middleware](#) to centralize communication between your services. And you will have various [Proxies](#), such as a [Firewall](#), a [Reverse Proxy](#), and a [Response Cache](#). You may even deploy a *Proxy* per kind of client if the clients vary in protocols, resulting in [Backends for Frontends \(BFF\)](#).



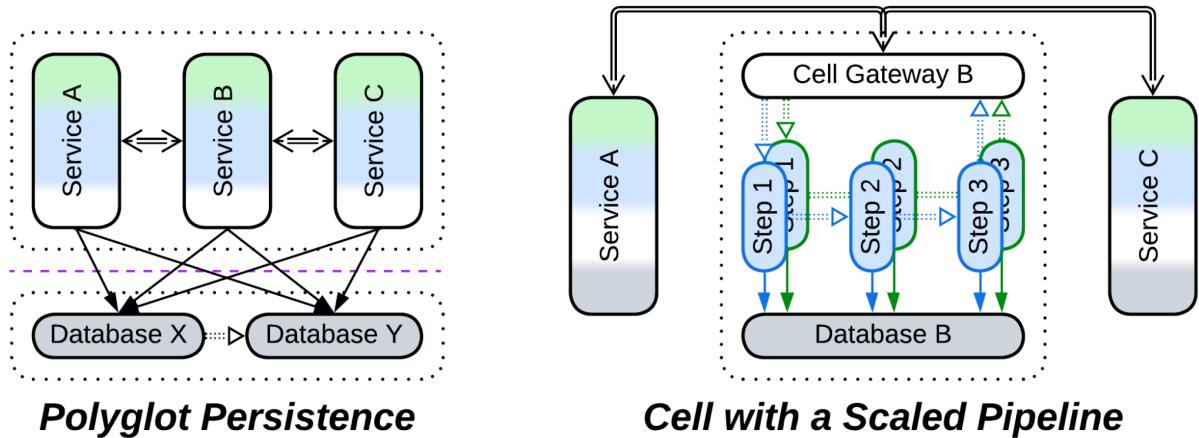
Runtime performance

Moreover, there are non-functional requirements, such as performance and fault tolerance.

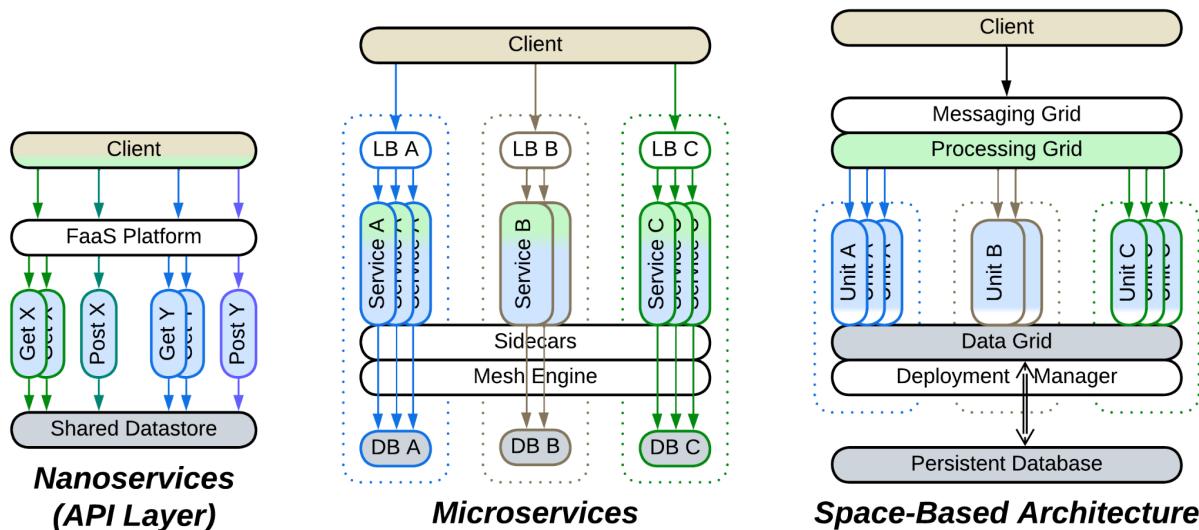
High throughput is achieved by [sharding](#) or [replicating](#) your business logic or even your data. Sharding also helps process huge datasets while replication improves fault tolerance. [Space-Based Architecture](#) replicates the entire dataset in memory for faster access.



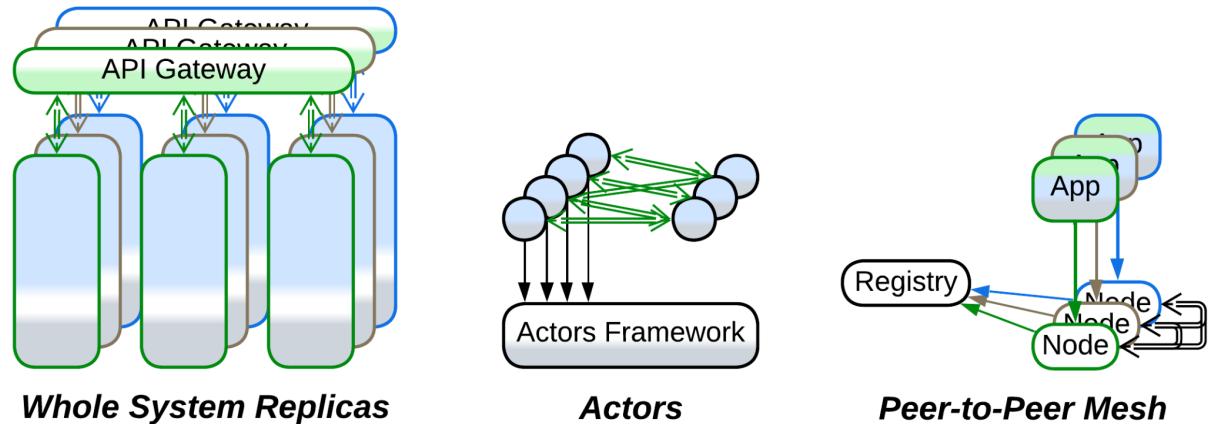
Alternatively, you may use several specialized databases ([Polyglot Persistence](#)) or redesign a highly loaded part of your system as a self-scaling [Pipeline](#).



Scalability under uneven load is achieved through [Function as a Service](#) (Nanoservices), [Service-Mesh](#)-based [Microservices](#) and, to a greater extent, [Space-Based Architecture](#).

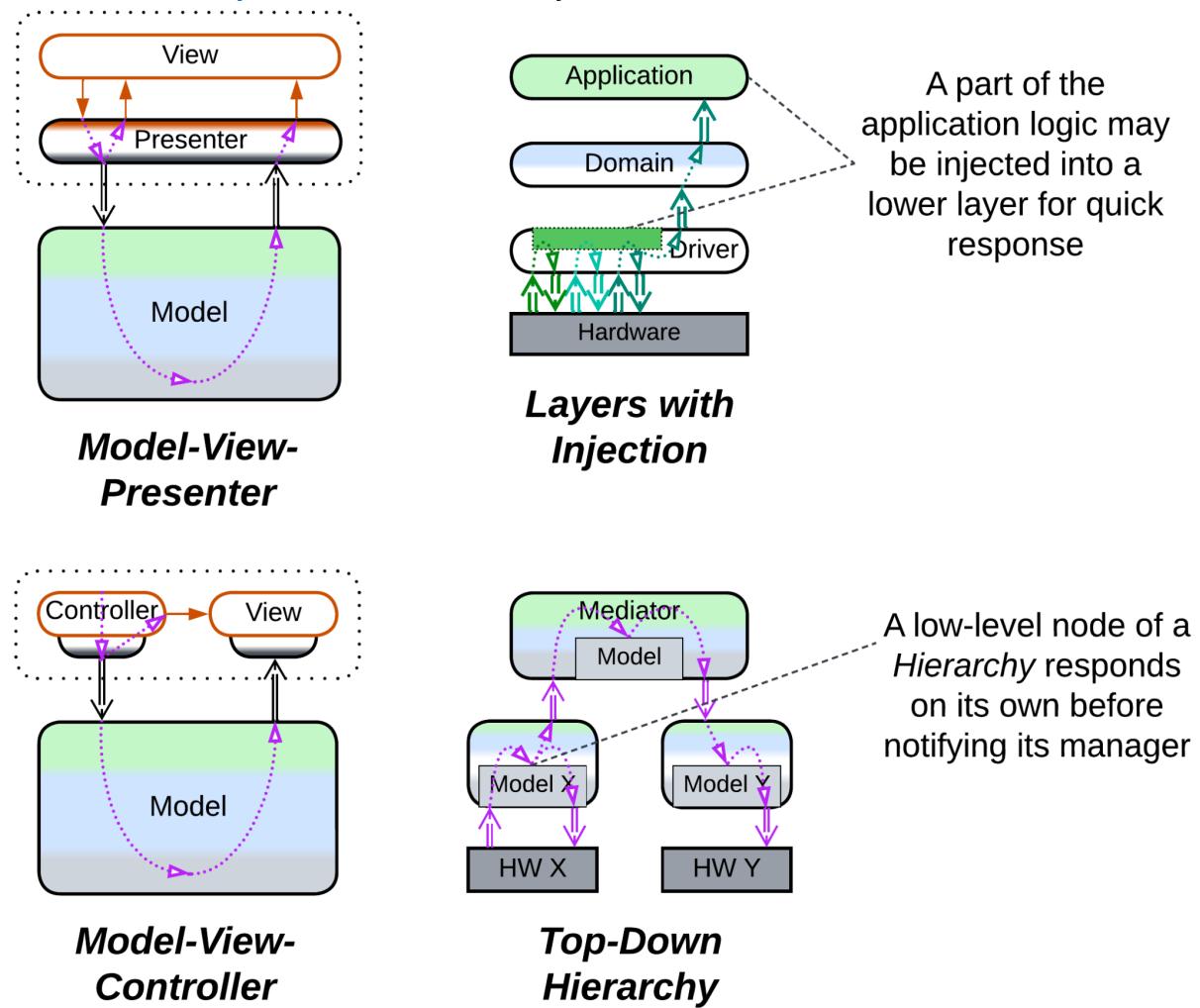


Fault tolerance requires you to have [replicas](#) of every component, including databases, ideally over multiple data centers. If you are not that rich, be content with [Actors](#) or [Mesh](#).



Low latency makes you place simplified first response logic close to your input, leading to:

- [Model-View-Presenter](#) or [Model-View-Controller](#) pattern families for user interaction.
- [Layers](#) with [strategy injection](#) for single hardware input.
- A [Hierarchy](#) for distributed [control](#) systems such as [IIoT](#).

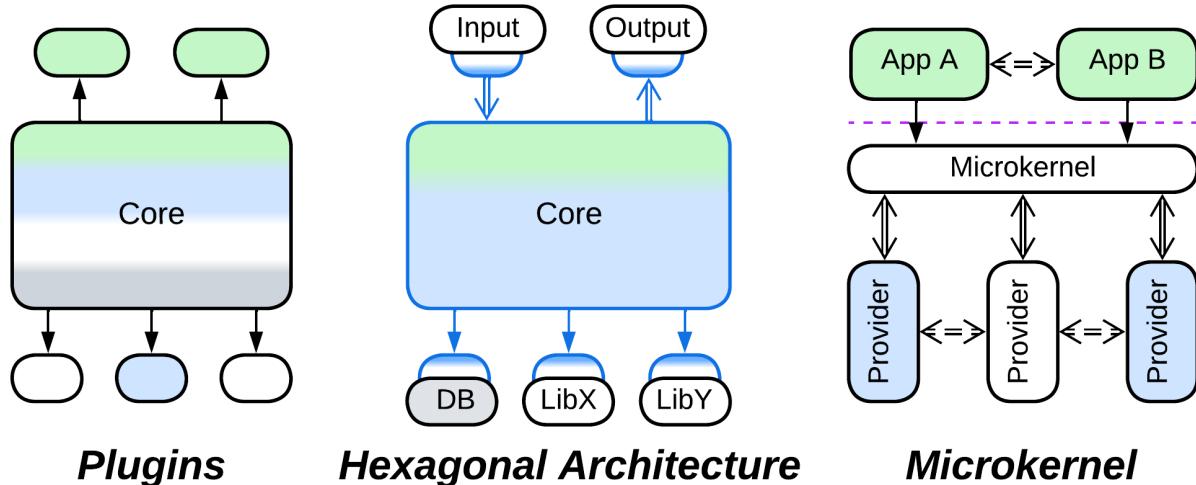


Flexibility

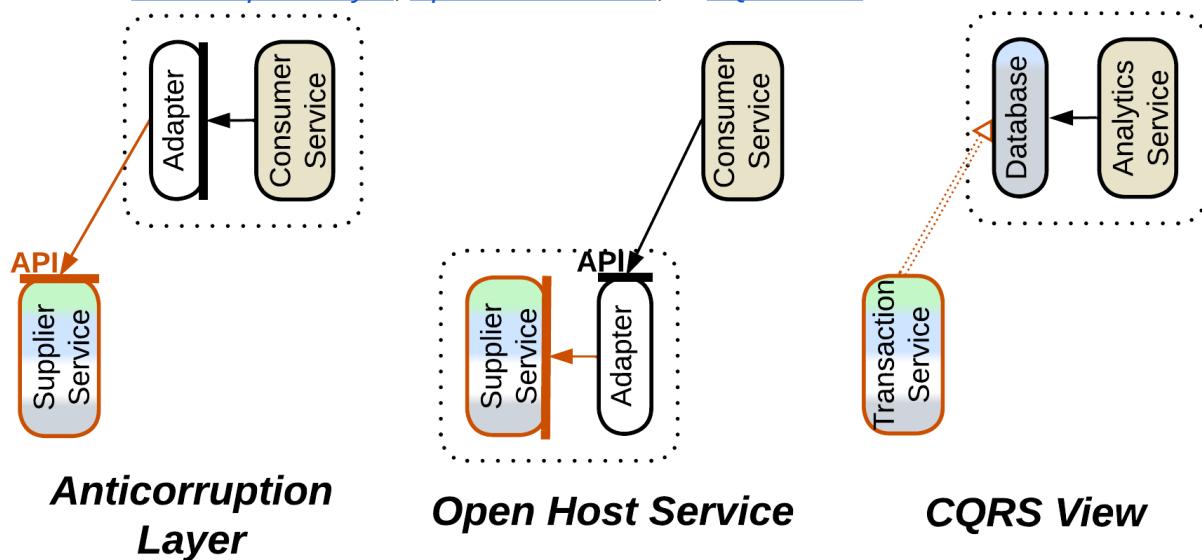
If your product needs customization, you go for [Plugins](#).

If it is to survive for a decade, you need [Hexagonal Architecture](#) to be able to swap vendors.

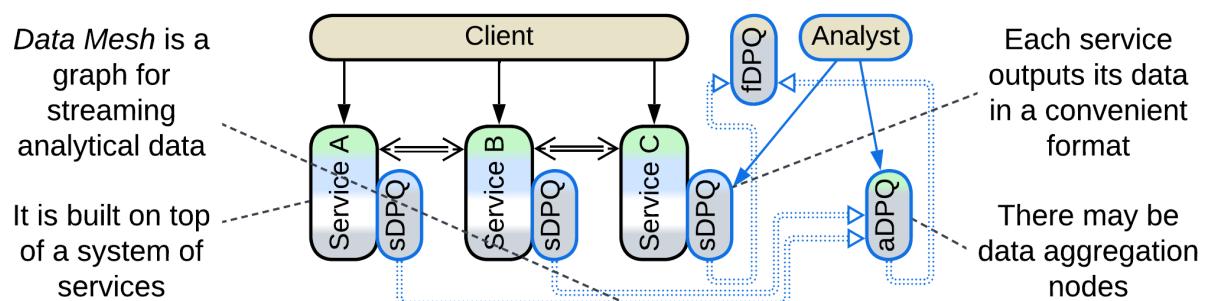
If you mediate between resource or service providers and consumers, you build a [Microkernel](#).



When your teams develop services and you want them to be [less interdependent](#), you insert an [Anticorruption Layer](#), [Open Host Service](#), or [CQRS View](#) between them.



When you have built a large system and really need that thorough data analytics, consider implementing a [Data Mesh](#).



Every domain is unique

No one-size-fits-all. Embedded projects or single-player games don't have databases and run in a single process. High Frequency Trading bypasses the OS kernel to save microseconds. *Middleware* and distributed databases care about quorum and leader election. Huge-scale data processing must account for [bit flips](#). A medical device should never crash. Banks store their history forever for external audits.

There is no universal architecture. No silver bullet pattern. Patterns are mere tools. Know your tools and choose wisely.

So it goes

Software architecture lies lifeless in my hands, devoid of its magical colors, *like the dead iguana*.

Part 7. Appendices

Appendix A. Acknowledgements.

I remember Avraham Fraenkel of DSPG who showed me what a true manager is like.

Thanks to Alexey Nikitin and Maxim Medvedev of Keenetic who let me design a subsystem from scratch and see how it fared through years of heavy changes.

It was from discussion with Sergey Ignatchenko aka [IT Hare](#) that I learned the difference between [control](#) and [data processing](#) systems. Mark Richards read my [previous series of articles](#) and encouraged me to press on with the classification of patterns. Kiarash Irandoust noticed [my articles on Medium](#) and invited me to publish them in [ITNEXT](#) where many more people could see them.

Thanks to Max Grom and other participants of the Ukrainian software architecture chat for hours of heated discussions about the meaning of patterns, which resulted in several analytical chapters of the book and for guiding me through the intricacies of DDD.

Many thanks to Lars Noodén for editing the entire book.

I must thank my mother and our neighbor Halyna for helping me throughout the war.

I want to thank everybody who prayed for me.

This book was made possible by many people who sacrificed their happy years, their limbs and their lives to protect those who stayed behind.

Appendix B. Books referenced.

DDD – Domain-Driven Design: Tackling Complexity in the Heart of Software. *Eric Evans*. Addison-Wesley (2003). (Most of these patterns are also well-described in [[LDDD](#)])

DDIA – Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann*. O'Reilly Media, Inc. (2017).

DDS – Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. *Brendan Burns*. O'Reilly Media, Inc. (2018).

DEDS – Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka. *Ben Stopford*. O'Reilly Media, Inc. (2018).

EIP – Enterprise Integration Patterns. *Gregor Hohpe and Bobby Woolf*. Addison-Wesley (2003).

FSA – Fundamentals of Software Architecture: An Engineering Approach. *Mark Richards and Neal Ford*. O'Reilly Media, Inc. (2020).

GoF – Design Patterns: Elements of Reusable Object-Oriented Software. *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*. Addison-Wesley (1994).

LDDD – Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. *Vlad Khononov*. O'Reilly Media, Inc. (2021). (Duplicates [[DDD](#)] thus I marked as [[LDDD](#)] only patterns not covered by [[DDD](#)])

MP – Microservices Patterns: With Examples in Java. *Chris Richardson*. Manning Publications (2018).

PEAA – Patterns of Enterprise Application Architecture. *Martin Fowler*. Addison-Wesley Professional (2002).

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).

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).

POSA3 – Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. *Michael Kircher, Prashant Jain*. John Wiley & Sons, Inc. (2004).

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).

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

SAHP – Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. *Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani. O'Reilly Media, Inc. (2021)*.

SAP – Software Architecture Patterns. *Mark Richards. O'Reilly Media, Inc. (2015)*.
(All of the architectures referenced here are in [[FSA](#)] as well, but [SAP] is free)

Appendix C. Copyright.

Attribution 4.0 International

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

Licensor means the individual(s) or entity(ies) granting rights under this Public License.

Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 – Scope.

License grant .

Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

reproduce and Share the Licensed Material, in whole or in part; and produce, reproduce, and Share Adapted Material.

Exceptions and Limitations . For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

Term . The term of this Public License is specified in Section 6(a) .

Media and formats; technical modifications allowed . The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

Downstream recipients .

Offer from the Licensor – Licensed Material . Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

No downstream restrictions . You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

No endorsement . Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i) .

Other rights .

Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

Patent and trademark rights are not licensed under this Public License.

To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

Attribution .

If You Share the Licensed Material (including in modified form), You must:

retain the following if it is supplied by the Licensor with the Licensed Material:

identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

a copyright notice;

a notice that refers to this Public License;
a notice that refers to the disclaimer of warranties;
a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

Sections 1 , 5 , 6 , 7 , and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Appendix D. Disclaimer.

THIS BOOK IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, FAILING AN INTERVIEW, BEING RIDICULED OR LOSING YOUR JOB) ARISING IN ANY WAY OUT OF THE USE OF THIS BOOK, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Appendix E. Evolutions.

This appendix details dozens of evolutions of metapatterns to show how they connect together. The evolutions probably have practical value through listing prerequisites, benefits, and drawbacks, but I am not sure that many readers will get through them without becoming bored to death. The metapattern chapters in the main parts of the book include abridged versions of the sections below.

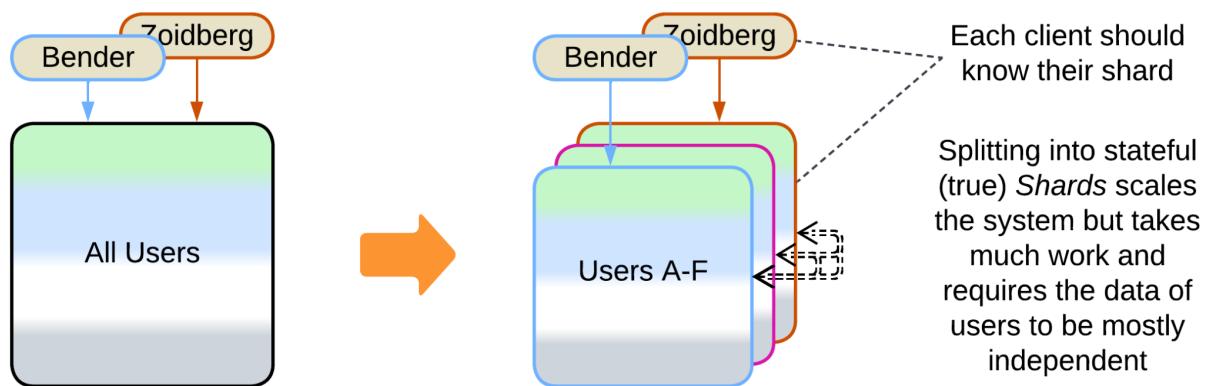
Duplicate and similar evolutions are omitted, and I did not write any evolutions for [fragmented metapatterns](#) as you should be able to infer them on your own after having read the book. Furthermore, for some reason I don't know of any evolutions for the [implementation metapatterns](#).

Monolith: to Shards

One of the main drawbacks of the monolithic architecture is its lack of scalability – a single running instance of your system may not be enough to serve all its clients no matter how many resources you add in. If that is the case, you should consider [Shards](#) – multiple instances of a monolith. There are following options:

- Self-managed [Shards](#) – each instance owns a part of the system's data and may communicate with all the other instances (forming a [Mesh](#)).
- [Shards with a Sharding Poxy](#) – each instance owns a part of the system's data and relies on the external component to choose a shard for a client.
- A [Pool of stateless instances](#) with a [Load Balancer](#) and a [Shared Database](#) – any instance can process any request, but the database limits the throughput.
- A [stateful instance per client](#) with an external persistent storage – each instance owns the data related to its client and runs in a virtual environment (i.e. web browser or an [actor framework](#)).

Implement a Mesh of self-managed shards



[Patterns: Sharding \(Shards\), Mesh.](#)

[Goal:](#) scale a low-latency application with weakly coupled data.

[Prerequisite:](#) the application's data can be split into semi-independent parts.

It is possible to run several instances of an application (*shards*), with each instance owning a part of the data. For example, a chat may deploy 16 servers, each responsible for a subset of users whose hashed names end in specific 4 bits (0 to 15). However, some scenarios (renaming a user, adding a contact) may require the shards to intercommunicate.

And the more coupled the shards become, the more complex a *mesh engine* is required to support their interactions, up to implementing distributed transactions, at that point you will have written a distributed database.

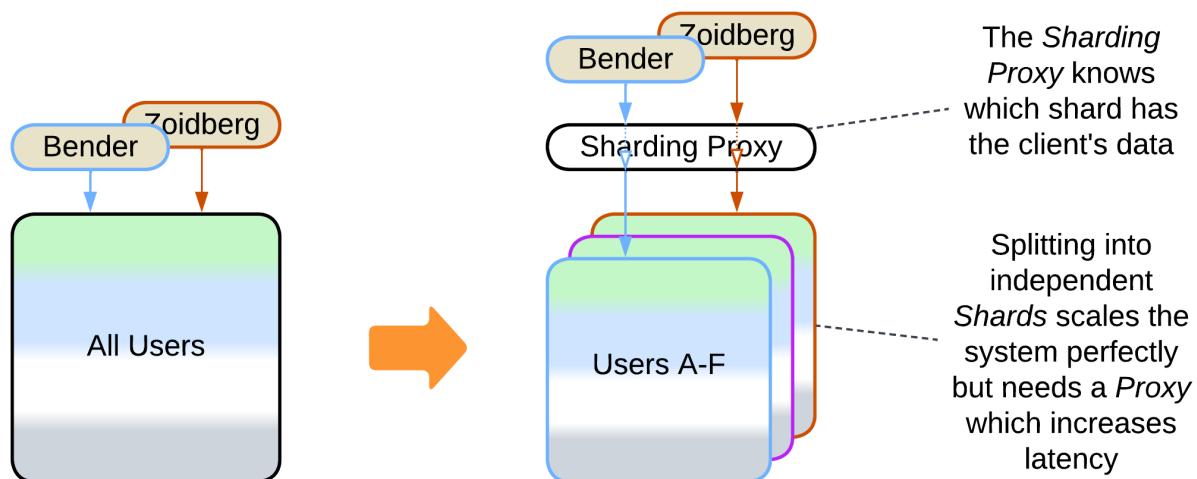
Pros:

- The system scales to a predefined number of instances.
- Perfect fault tolerance if [replication](#) and error recovery are implemented.
- Latency is kept low.

Cons:

- Direct communication between shards (the mesh engine logic) is likely to be quite complex.
- Intershard transactions are slow and/or complicated and may corrupt data if undetected.
- A client must know which shards own its data to benefit from low latency. An [Ambassador Sharding Proxy](#) may be used on the client's side.

Split data to isolated shards and add a Sharding Proxy



Patterns: [Sharding \(Shards\)](#), [Sharding Proxy \(Proxy\)](#), [Layers](#).

Goal: scale an application with sliceable data.

Prerequisite: the application's data can be sliced into independent, self-sufficient parts.

If all the data a user operates on, directly or indirectly, is never accessed by other users, then multiple independent instances (*shards*) of the application can be deployed, each owning an instance of a database. A special kind of *Proxy*, called *Sharding Proxy*, redirects a user request to a shard that has the user's data.

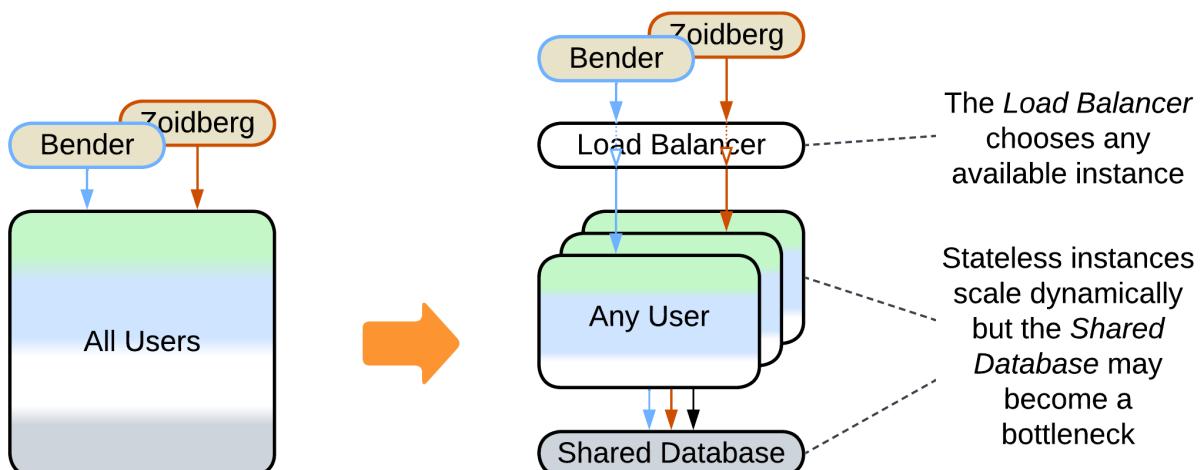
Pros:

- Perfect static (predefined number of instances) scalability.
- Failure of a shard does not affect users of other shards.
- [Canary Release](#) is supported.

Cons:

- The *Sharding Proxy* is a single point of failure unless [replicated](#) and increases latency unless deployed as an [Ambassador \[DDS\]](#).

Separate the data layer and add a load balancer



Patterns: [Pool \(Shards\)](#), [Shared Database \(Shared Repository\)](#), [Load Balancer \(Proxy\)](#), [Layers](#).

Goal: achieve scalability with little effort.

Prerequisite: there is persistent data of manageable size.

As data moves into a dedicated layer, the application becomes stateless and instances of it can be created and destroyed dynamically depending on the load. However, the *Shared Database* becomes the system's bottleneck unless [Space-Based Architecture](#) is used.

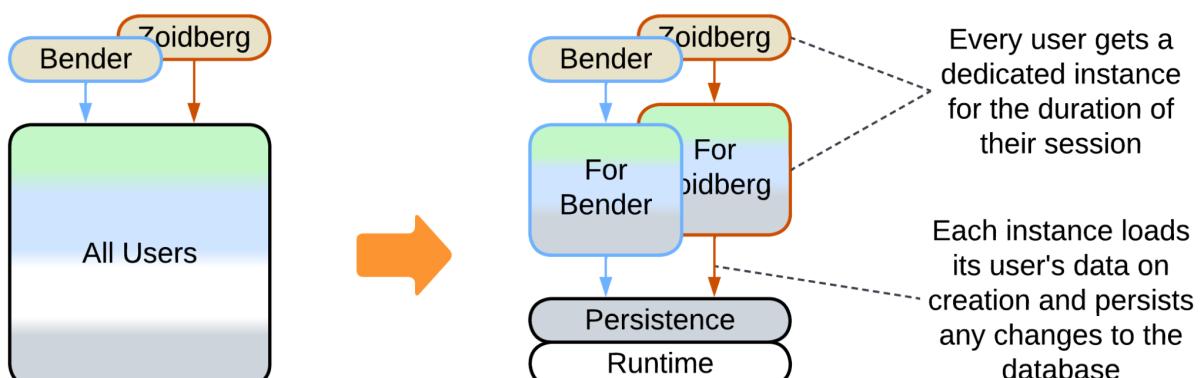
Pros:

- Easy to implement.
- Dynamic scalability.
- Failure of a single instance affects few users.
- [Canary Release](#) is supported.

Cons:

- The database limits the system's scalability and performance.
- The *Load Balancer* and *Shared Database* increase latency and are single points of failure.

Dedicate an instance to each client



Patterns: [Create on Demand \(Shards\)](#), [Shared Repository](#), [Virtualizer \(Microkernel\)](#), [Layers](#).

Goal: very low latency, dynamic scalability, and failure isolation.

Prerequisite: each client's data is small and independent of other clients.

Each client gets an instance of the application which preloads their data into memory. This way all the data is instantly accessible and a processing fault from one client never affects the other clients. As systems tend to have thousands to millions of clients, it is inefficient to spawn a process per client. Instead, more lightweight entities are used: a web app in a browser or an *actor* in a [distributed framework](#).

Pros:

- Nearly perfect dynamic scalability (limited by the persistence layer).
- Good latency as everything happens in RAM.
- Fault isolation is one of the features of distributed frameworks.
- Frameworks are available out of the box.

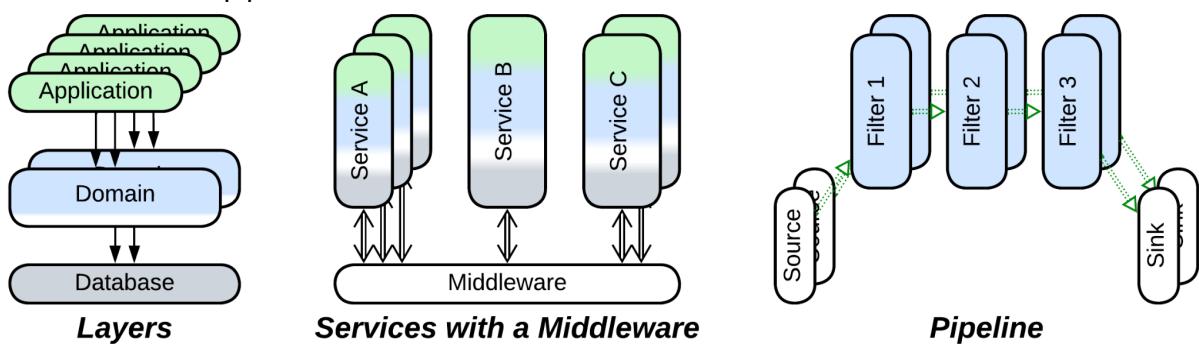
Cons:

- Virtualization frameworks tend to introduce a performance penalty.
- You may need to learn an uncommon technology.
- Scalability and performance are still limited by the shared persistence layer.

Further steps

In most cases *sharding* does not change much inside the application, thus the common evolutions for [Monolith](#) (to [Layers](#), [Services](#), and [Pipeline](#)) remain applicable after sharding. We'll focus on their scalability:

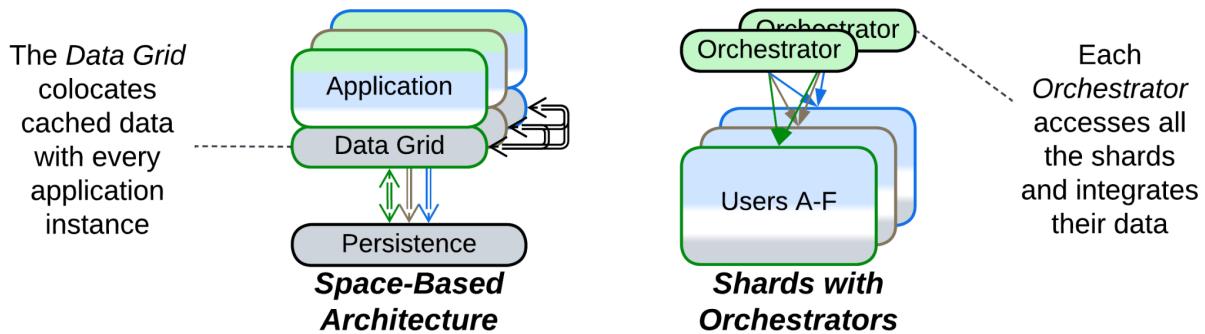
- [Layers](#) can be scaled (often to a dramatic extent) and deployed individually, as exemplified by the [Three-Tier Architecture](#).
- [Services](#) allow for subdomains to scale independently with the help of [Load Balancers](#) or a [Middleware](#). They also improve performance of data storage as each service uses its own database which is often chosen to best fit its distinct needs.
- Granular scaling can apply to [Pipelines](#), but in many cases that does not make much sense as pipeline components tend to be lightweight and stateless, making it easy to scale the pipeline as a whole.



There are specific evolutions of [Shards](#) that deal with their drawbacks:

- [Space-Based Architecture](#) reimplements [Shared Repository](#) with [Mesh](#). Its main goal is to make the data layer dynamically scalable, but the exact results are limited by the [CAP theorem](#) thus, depending on the mode of action, it can provide very high performance with no consistency guarantees for a small dataset, or reasonable performance for a huge dataset. It blends the best features of stateful [Shards](#) and [Shared Database](#) (being an option for either to evolve to) but may be quite expensive to run and lacks algorithmic support for analytical queries.
- [Orchestrator](#) is a mirror image of [Shared Database](#), another option to implement use cases that deal with data of multiple shards without the need for the shards to

intercommunicate. Stateless *Orchestrators* scale perfectly but may corrupt the data if two of them write to an overlapping set of records.

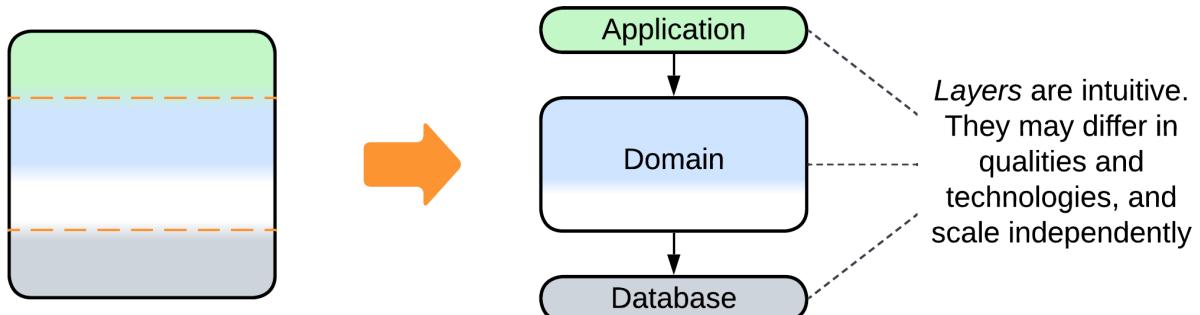


Monolith: to Layers

Another drawback of [Monolith](#) is its ... er ... monolithic nature. The entire application exposes a single set of qualities and all its parts (if they ever emerge) are deployed together. However, life awards flexibility: parts of a system may benefit from being written in varying languages and styles, deployed with different frequency and amount of testing, sometimes to specific hardware or end users' devices. They may need to [vary in security and scalability](#) as well. Enter [Layers](#) – a subdivision by the *level of abstractness*:

- Most *Monoliths* can be divided into three or four [layers](#).
- It is common to see the database separated from the main application.
- [Proxies](#) (e.g. [Firewall](#), [Cache](#), [Reverse Proxy](#)) are common additions to the system.
- An [Orchestrator](#) adds a layer of indirection to simplify the system's API for its clients.

Divide into Layers



Patterns: [Layers](#).

Goal: let parts of the system vary in qualities, improve the structure of the code.

Prerequisite: there is a natural way to separate the high-level logic from the low level implementation details and dependencies.

Most systems apply *layering* by default as it grants a lot of flexibility at very little cost.

[Common sets of layers](#) are: UI, tasks (orchestration), domain (detailed business rules) and infrastructure (database and libraries) or frontend, backend and data.

Pros:

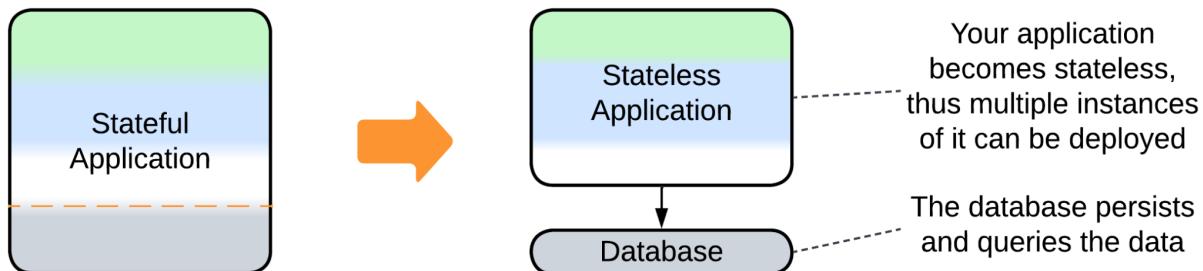
- It is a natural way to specialize and decouple two or three development teams.
- The layers may vary in virtually any quality:
 - They are deployed and scaled independently.
 - They may run on different hardware, including client devices.
 - They may vary in programming language, paradigm and release cycle.

- Most changes are isolated to a single layer.
- Layering opens a way to many evolutions of the system.
- The code [becomes easier to read](#).

Cons:

- Dividing an existing application into *Layers* may take some effort.
- There is a small performance penalty.

Use a database



Patterns: [Layers](#), [Shared Database \(Shared Repository\)](#).

Goal: avoid implementing a datastore.

Prerequisite: the system needs to query (maybe also persist) a large amount of data.

A datastore is non-trivial to implement. While ordinary files are good for small volumes of data, as your needs grow so needs to grow your technology. Deploy a database.

Pros:

- A well-known database is sure to be more reliable than any in-house implementation.
- Many databases provide heavily optimized algorithms for querying data.
- You can choose other hardware to deploy the database to.
- Your (now stateless) application will be easy to scale.

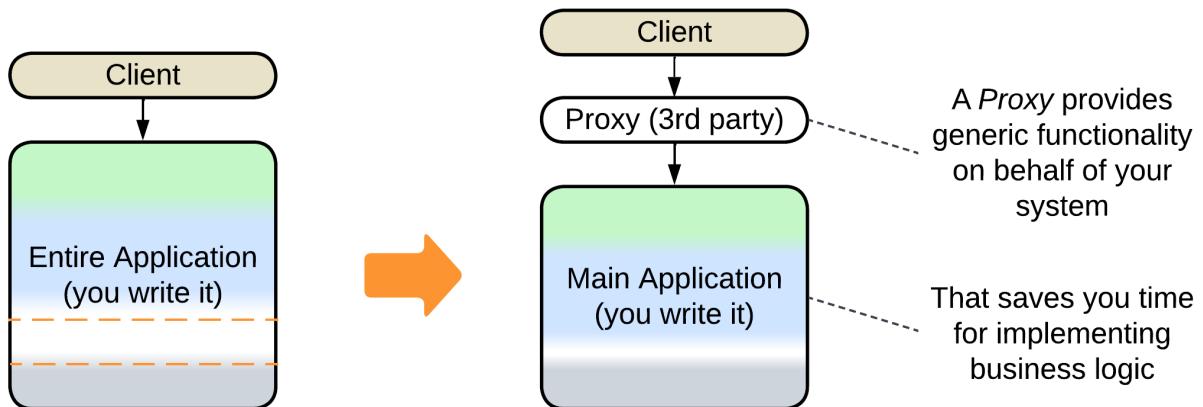
Cons:

- Databases are complex and require fine-tuning.
- You cannot adapt the database engine to your evolving needs.
- Most databases do not scale.
- You are stepping right into [vendor lock-in](#).

Further steps:

- Deploy multiple [instances](#) of your application behind a [Load Balancer](#).
- Continue the transition to [Layers](#) by separating the high-level and low-level business logic.
- [Polyglot Persistence](#) improves performance of the data layer.
- [CQRS](#) passes read and write requests through dedicated services.
- [Space-Based Architecture](#) is low latency and allows for dynamic scalability of the whole system, including the data layer.
- [Hexagonal Architecture](#) will allow you to switch to another database in the future.

Add a Proxy



Patterns: [Layers](#), [Proxy](#).

Goal: avoid implementing generic functionality.

Prerequisite: Your system serves clients (as opposed to [controlling hardware](#)).

A *Proxy* is placed between your system and its clients to provide generic functionality that otherwise would have to be implemented by the system. The kinds of *Proxy* to use with [Monolith](#) are: [Firewall](#), [Cache](#), [Reverse Proxy](#), and [Adapter](#). Multiple *Proxies* can be deployed.

Pros:

- You save some time (and money) on development.
- A well-known *Proxy* is likely to be more secure and reliable than an in-house implementation.
- You can select hardware to deploy the *Proxy* to.

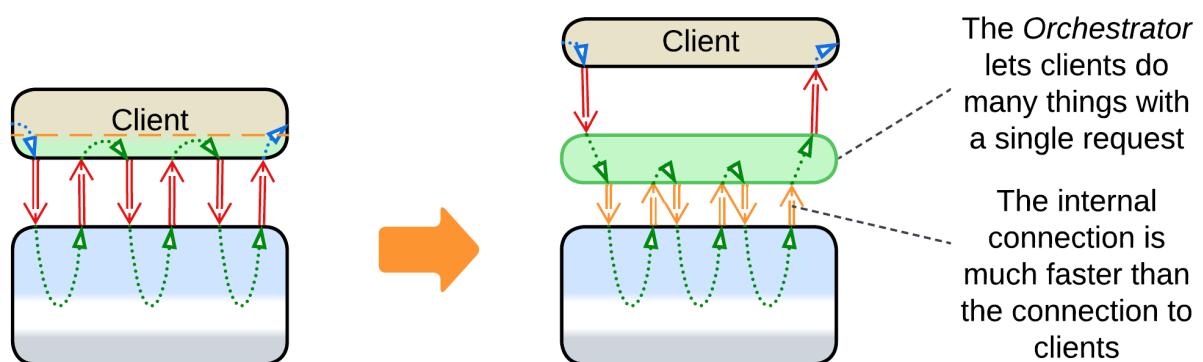
Cons:

- Latency degrades, except for [Response Cache](#) where it depends on frequency of requests.
- The *Proxy* may fail, which increases the chance of failure of your system.
- Beware of [vendor lock-in](#).

Further steps:

- Another kind of *Proxy* may be added.
- Some systems employ a *Proxy* per client, leading to [Backends for Frontends](#).

Add an Orchestrator



Patterns: [Layers](#), [Orchestrator](#).

Goal: provide a high-level API for clients to improve their developer experience and performance.

Prerequisite: the API of your system is fine-grained and there are common use cases which repeat certain sequences of calls to your API.

A well-designed *Orchestrator* should provide a high-level API which is intuitive, easy to use, and coarse-grained to minimize the number of interactions between the system and its clients. An old way to access the original system's API may still be maintained for rare use cases or legacy client applications ([open orchestration](#)). As a matter of fact, you program common logic on behalf of your clients.

Pros:

- Client applications become easier to write.
- Latency improves.

Cons:

- You get yet another moving part to design, test, deploy, and observe; and lots of meetings between development teams for a bonus.
- The new coarse-grained interface will likely be less powerful than the original one.

Further steps:

- [Backends for Frontends](#) use an *Orchestrator* per client type.

Further steps

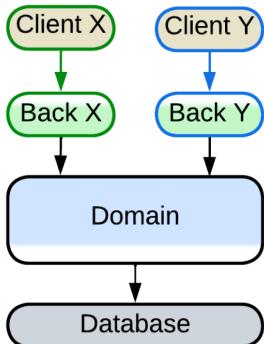
Applying one of the evolutions discussed above does not prevent you from following another one of them, or even the same one for a second time:

- A layer can be split into two layers.
- A database can be added.
- Multiple kinds of *Proxies* are OK.
- If you don't have an *orchestration layer* yet, you may add one.

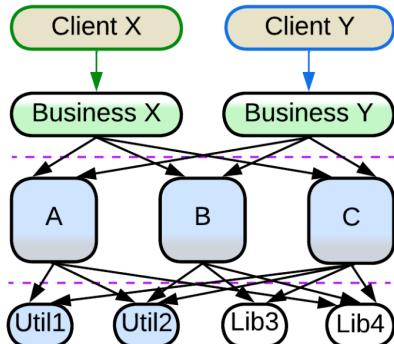
Those were evolutions from *Layers* to [Layers](#).

Another set of evolutions stems from splitting one or more *layers* into [Services](#):

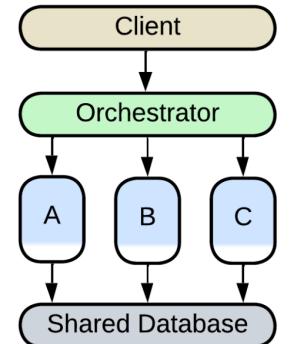
- Splitting a [Proxy](#) and/or [Orchestrator](#) yields [Backends for Frontends](#) where requests from each kind of client are processed by a dedicated component.
- Splitting the layer with the main business logic results in [Services](#), possibly augmented with layers of [Middleware](#), [Shared Database](#), [Proxies](#) and/or [Orchestrator](#).
- Splitting the *database layer* leads to [Polyglot Persistence](#) with specialized storages.
- If all the layers share the domain dimension and are split along it, [Layered Services](#) (or its subtype [CQRS](#)) emerge.
- If each layer is split along its own domain, the system follows [Service-Oriented Architecture](#) that is built around component reuse.
- Finally, some domains support [Hierarchy](#) – a tree-like architecture where each layer takes a share of the system's functionality.



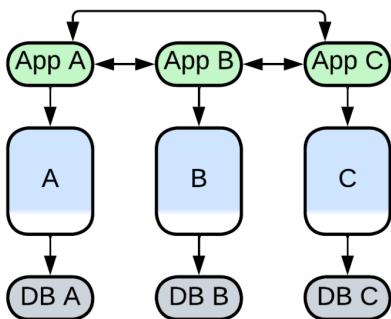
Backends for Frontends



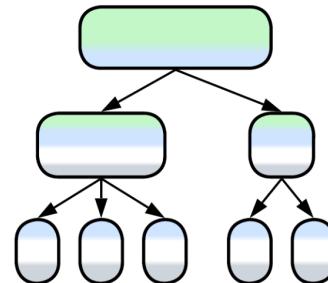
Service-Oriented Architecture



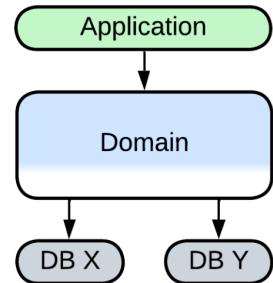
Services with an Orchestrator and a Shared Database



Layered Services



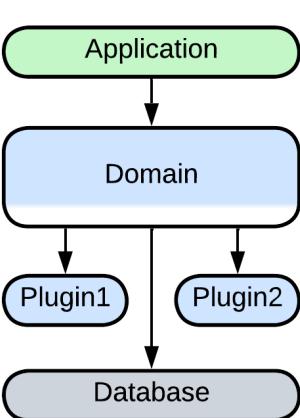
Hierarchy



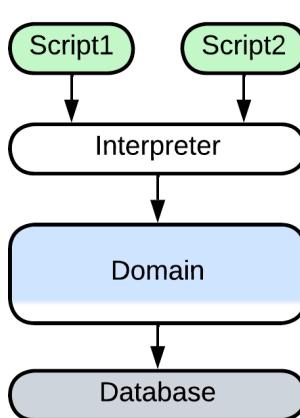
Polyglot Persistence

In addition,

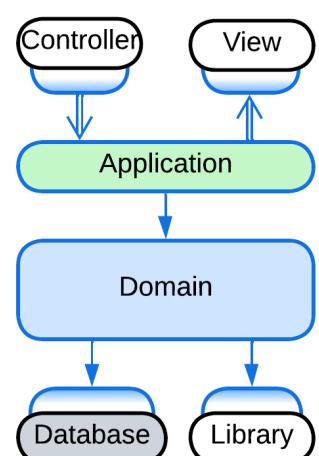
- Distributed systems usually allow for the [scaling](#) of one or more layers.
- A layer may employ [Plugins](#) for better customizability.
- The UI and infrastructure layers may be split and abstracted according to the rules of [Hexagonal Architecture](#) (or its subtype [Separated Presentation](#)).
- The system can often be extended with [Scripts](#), resulting in a kind of [Microkernel](#).



Layers with Plugins



Layers with Scripts

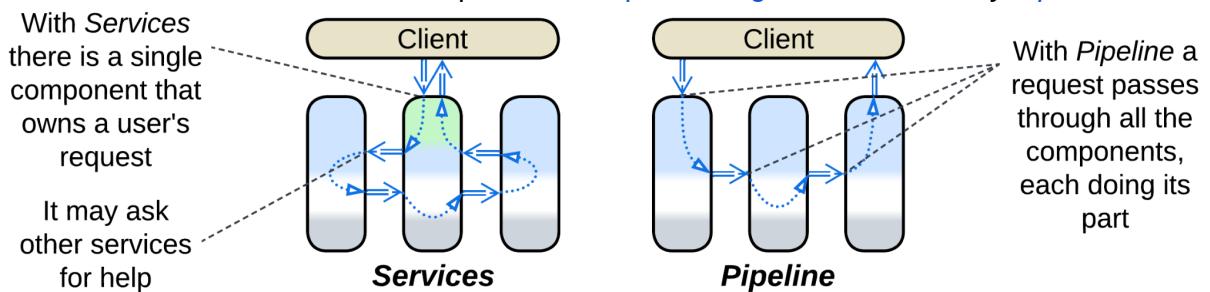


Layered Hexagonal Architecture

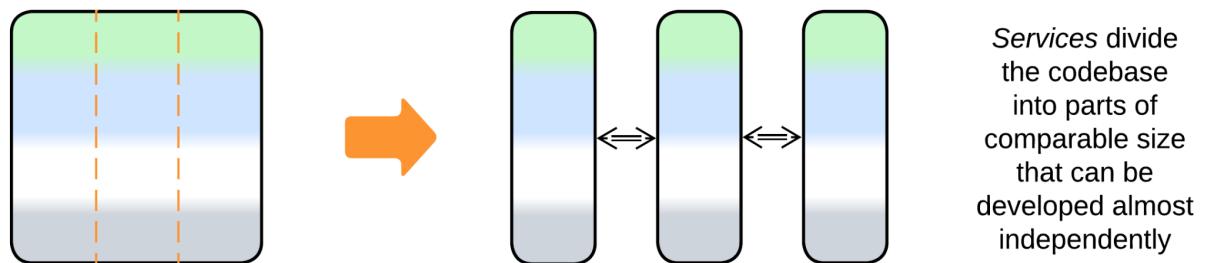
Monolith: to Services

The final major drawback of [Monolith](#) is the cohesiveness of its code. The rapid start of development begets a major obstacle for project growth: every developer needs to know the entire codebase to be productive and changes made by individual developers overlap and may break each other. Such distress is usually solved by dividing the project into components along *subdomain boundaries* (which tend to match [bounded contexts](#) [DDD]). However, that requires a lot of work, and good boundaries and APIs are [hard to design](#). Thus many organizations prefer a slower iterative transition.

- A *Monolith* can be split into [Services](#) right away.
- Or only the new features may be added as new services.
- Or the weakly coupled parts of existing functionality may be separated, one at a time.
- Some domains allow for sequential [data processing](#) best described by [Pipelines](#).



Divide into Services



Patterns: [Services](#).

Goal: facilitate development by multiple teams, improve the code, decouple qualities of subdomains.

Prerequisite: there is a natural way to split the business logic into loosely coupled subdomains, and the subdomain boundaries are sure to never change in the future.

Splitting a *Monolith* into *Services* by subdomain [is risky in the early stages of a project](#) while the domain understanding is evolving ([in-process modules](#) are less risky but provide fewer benefits). However, this is the way to go as soon as the codebase becomes unwieldy due to its size.

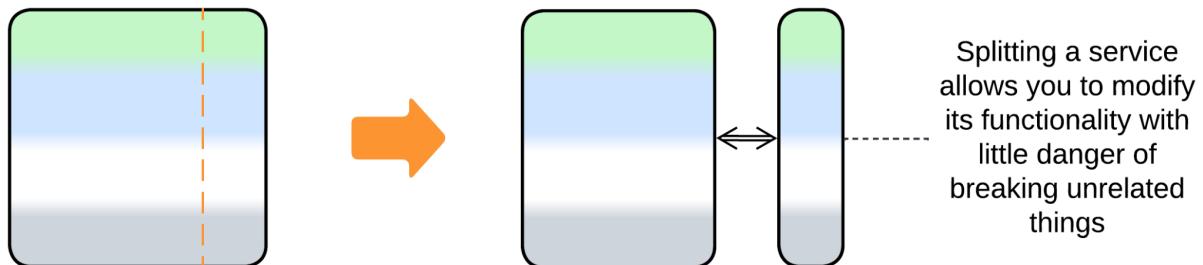
Pros:

- Supports multiple, relatively independent and specialized development teams.
- Lowers the penalty imposed by the project's size and complexity on the velocity of development and product quality.
- Each team may choose the best fitting technologies for its service.
- The services can differ in [non-functional requirements](#).
- Flexible deployment and scaling.
- A certain degree of error tolerance for asynchronous systems.

Cons:

- It takes lots of work to split a *Monolith*.
- Any future changes to the overall structure of the domain will be hard to implement.
- Sharing data between services is complicated and error-prone.
- System-wide use cases are hard to understand and debug.
- There is a moderate performance penalty for system-wide use cases.

Add or split a service



Patterns: [Services](#).

Goal: [stop digging](#), get some work for novices who don't know the entire project.

Prerequisite: the new functionality you are adding or the part you are splitting is weakly coupled to the bulk of the existing *Monolith*.

If your *Monolith* is already hard to manage, but a new functionality is needed, you can try dedicating a separate service to the new feature(s). This way the *Monolith* does not become larger – it is even possible that you will move a part of its code to the newly established service.

If you are not adding a new feature but need to change an old one – use the chance to make the existing *Monolith* smaller by first separating the functionality which you are going to change from its bulk. At the very minimum this two-step process lowers the probability of breaking something unrelated to the changes of behavior required.

Pros:

- The legacy code does not increase in size and complexity.
- The new service is transferred to a dedicated team which does not need to know the legacy system.
- The new service can be experimented with and even rewritten from scratch.
- The likely faults of the new service don't crash the main application.
- The new service can be tested and deployed in isolation.
- The new service can be scaled independently.

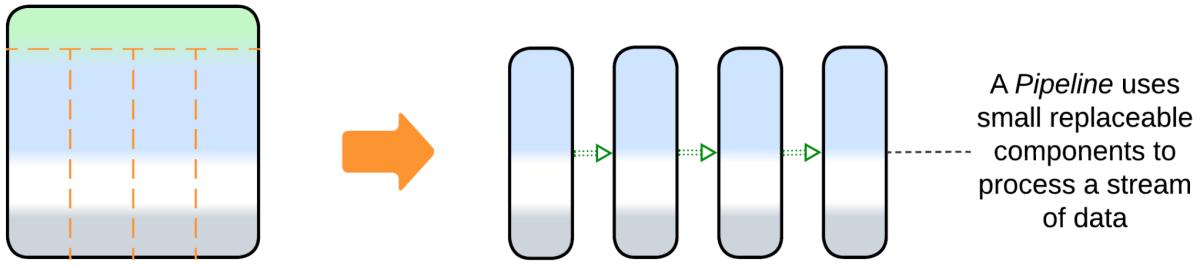
Cons:

- The new service will have a hard time sharing data or code with the main application.
- Use cases that involve both the new service and the old application are hard to debug.
- There is a moderate performance penalty for using the service.

Further steps:

- [Continue disassembling](#) the *Monolith*.

Divide into a Pipeline



Patterns: [Pipeline \(Services\)](#).

Goal: decrease the complexity of the code, make it easy to experiment with the steps of data processing, distribute the task over multiple CPU cores, processors or computers.

Prerequisite: the domain can be represented as a sequence of coarse-grained data processing steps.

If you can treat your application as a chain of independent steps that transform the input data, you can rely on the OS to schedule them and you can also dedicate a development team to each of the steps. This is the default solution for a system that [processes a stream](#) of a single type of data (video, audio, measurements). It has excellent flexibility.

Pros:

- Nearly abolishes the influence of project size on development velocity.
- The project's teams become almost independent.
- Flexible deployment and scaling.
- Naturally supports event replay for reproducing bugs, testing or benchmarking individual components.
- It is possible to have multiple implementations of each of the steps of data processing.
- Does not need any manual scheduling or thread synchronization.

Cons:

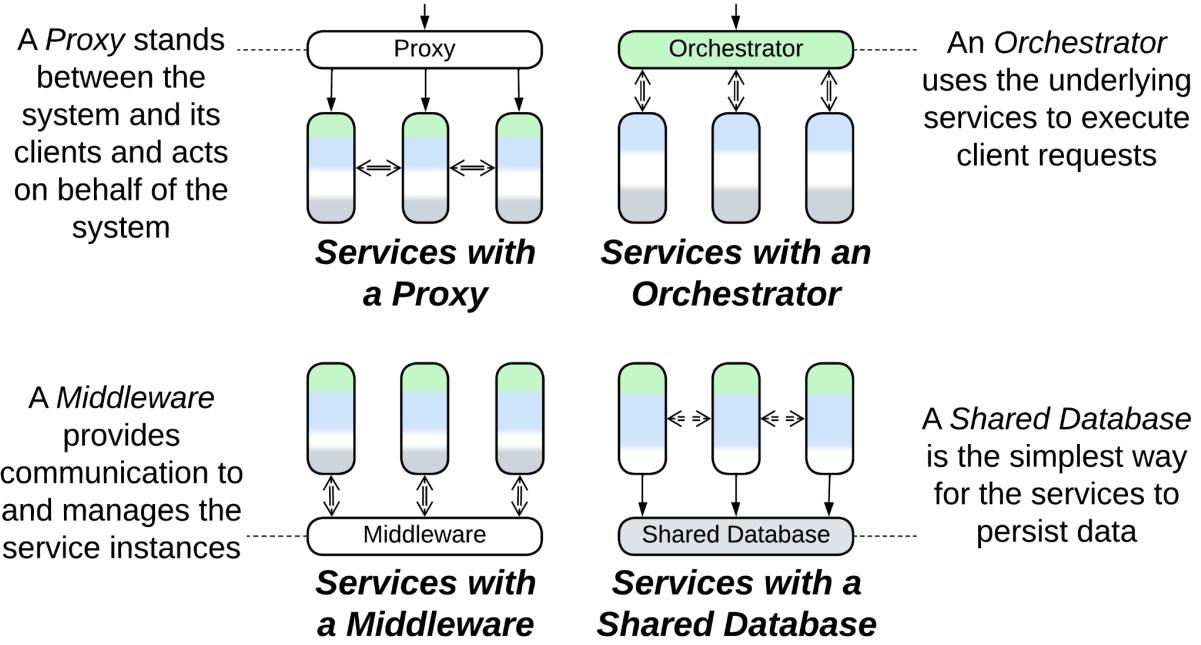
- Latency may skyrocket.
- As the number of supported scenarios grows, so does the number of components and *pipelines*. Soon there'll be nobody who understands the system as a whole.

Further steps

As your knowledge of the domain and your business requirements change, you may need to move some functionality between the services to keep them loosely coupled. Sometimes you have to merge two or three services together. So it goes.

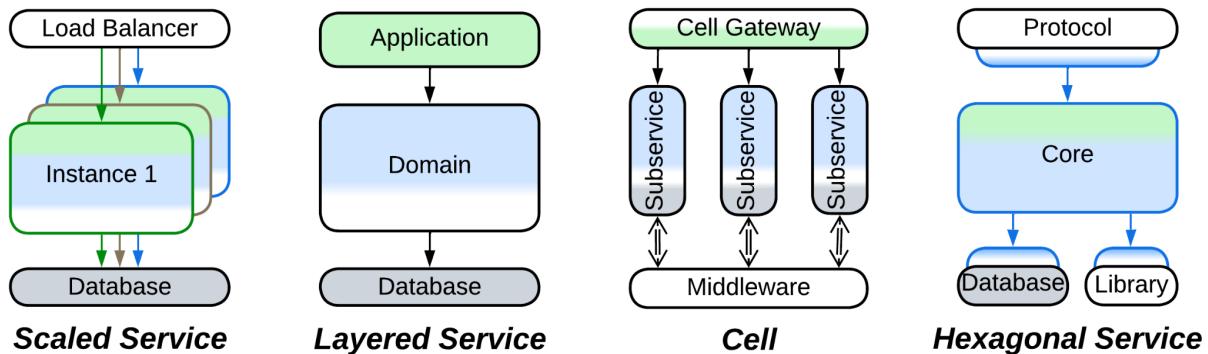
Systems of [Services](#) or [Pipelines](#) are quite often extended with special kinds of [layers](#):

- [Middleware](#) takes care of deployment, intercommunication and scaling of services.
- [Shared Repository](#) lets services operate on and communicate through [shared data](#).
- [Proxies](#) are ready-to-use components that add generic functionality to the system.
- [Orchestrator](#) encapsulates use cases that involve multiple services, so that the services don't need to know about each other.
- Finally, there are [Combined Components](#) that implement two or more of the above patterns in a single framework.



Each service, being a smaller *Monolith*, may evolve on its own. Most of the evolutions of [*Monolith*](#) are applicable. The most common examples include:

- [*Scaled \(Sharded\) Service*](#) with a [*Load Balancer*](#) and [*Shared Database*](#) to support high load.
- [*Layered Service*](#) to improve the code structure and decouple deployment of parts of a service.
- [*Cell* \(Service of Services\)](#) to involve multiple teams and technologies within a single subdomain.
- [*Hexagonal Service*](#) to escape vendor lock-in.

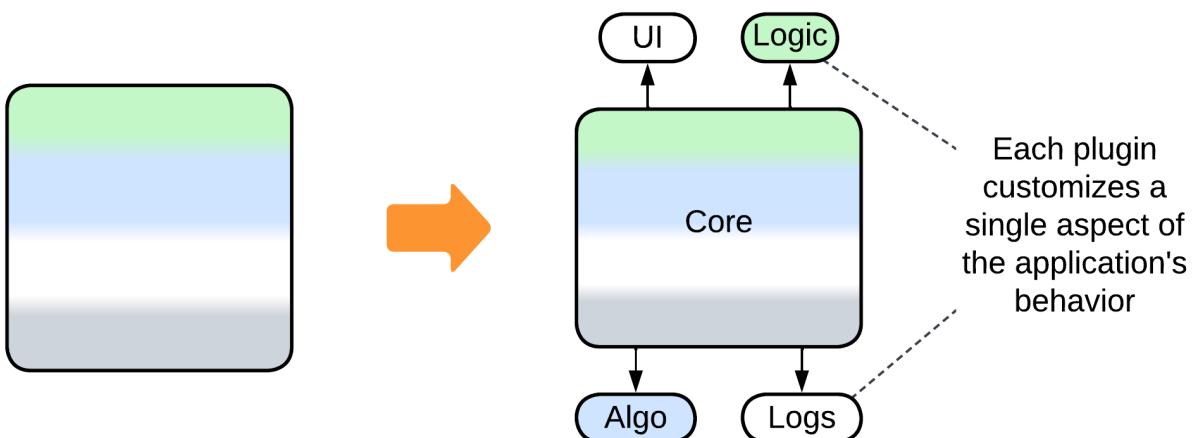


Monolith: to Plugins

The last group of evolutions which we review does not really change the monolithic nature of the application. Instead, its goal is to improve the *customizability* of the [*Monolith*](#):

- Vanilla [*Plugins*](#) is the most direct approach which relies on replaceable bits of logic.
- [*Hexagonal Architecture*](#) is a subtype of *Plugins* which is all about isolating the main code from any third-party components it uses.
- [*Scripts*](#) is a kind of [*Microkernel*](#) – yet another subtype of *Plugins* – which gives users of the system full control over its behavior.

Support plugins



Patterns: [Plugins](#).

Goal: simplify the customization of the application's behavior.

Prerequisite: several aspects need to vary from customer to customer.

Plugins create points of access to the system that allow engineers to collect data and govern select aspects of the system's behavior without having to learn the system's implementation.

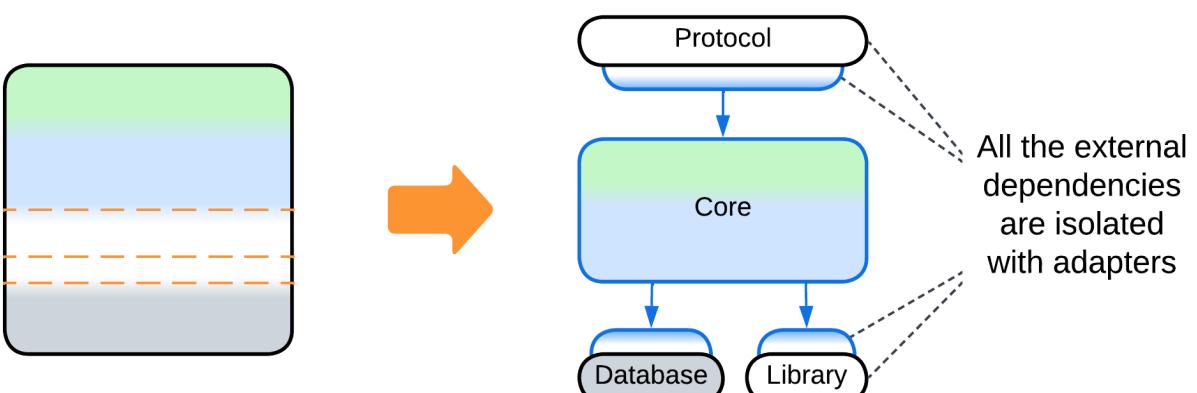
Pros:

- The system can be modified by internal and external programmers who don't know its internal details.
- Customized versions become much easier to release and support.

Cons:

- Extensive changes may be required to expose the tunable aspects of the system.
- Testability becomes poor because of the number of possible variants.
- Performance is likely to degrade.

Isolate dependencies with Hexagonal Architecture



Patterns: [Hexagonal Architecture \(Plugins\)](#).

Goal: isolate the business logic from external dependencies.

Prerequisite: there are third-party or unstable components in the system.

The main business logic will communicate with all the external components through APIs or SPIs defined in the terms of the business logic. This way it will not depend on anything at all and any component will be replaceable with another implementation or a [stub/mock](#).

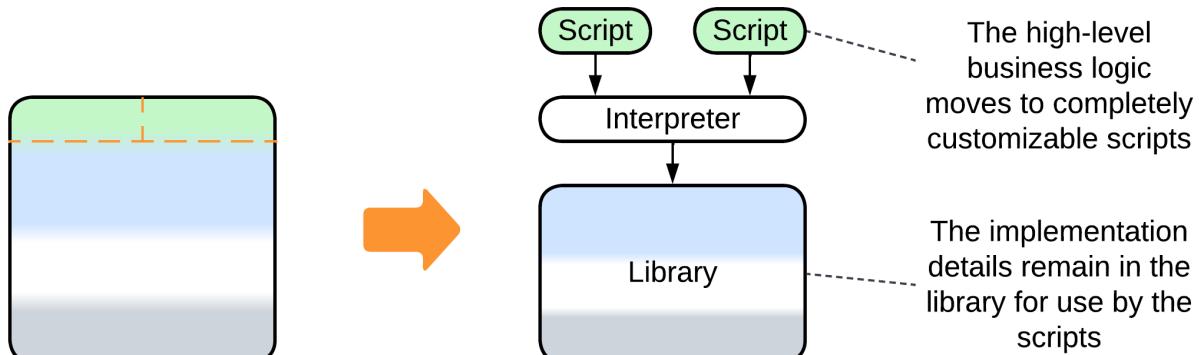
Pros:

- Vendor lock-in is ruled out.
- A component may be replaced through to the very end of the system's life cycle.
- Stubs and mocks are supported for testing and local or early development.
- It is possible to provide multiple implementations of a component.

Cons:

- Some extra effort is required to define and use the interfaces.
- There is performance degradation, mostly due to lost optimization opportunities.

Add an Interpreter (support Scripts)



Patterns: [Scripts aka Interpreter](#) ([Microkernel \(Plugins\)](#)).

Goal: allow the system's users to implement their own business logic.

Prerequisite: the domain is representable in high-level terms.

Interpreter lets the users develop high-level business logic from scratch by programming interactions of pre-defined building blocks which are implemented in the core of the system. That provides unparalleled flexibility at the cost of performance and design complexity.

Pros:

- Perfect flexibility and customizability for every user.
- The high-level business logic is written in high-level terms, making it fast to develop and easy to grasp.

Cons:

- Requires much effort to design correctly.
- There may be a heavy performance penalty if the API is too fine-grained.
- Testability may be an issue.

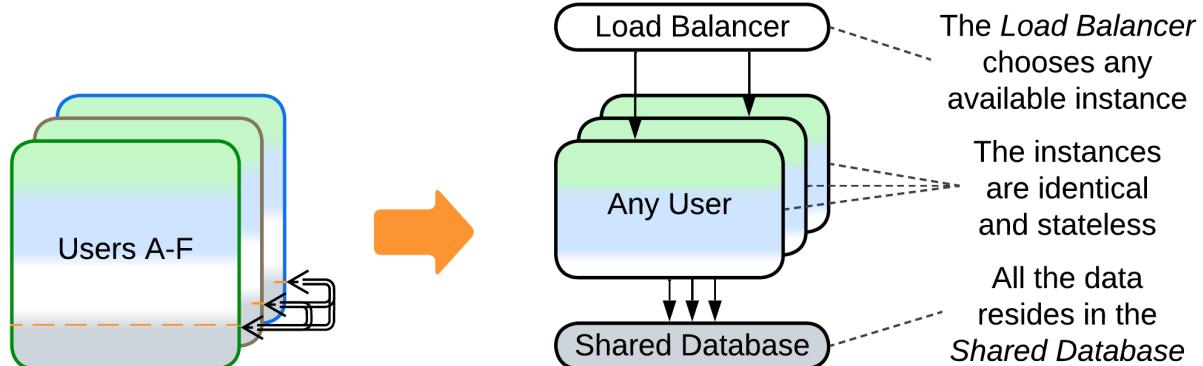
Shards: share data

One issue peculiar to [Shards](#) is that of coordinating the instances deployed, especially if their data become coupled. The most direct solution is to let the instances operate a component that wraps the shared data:

- If the whole dataset needs to be shared, it can be split into a [Shared Repository](#) layer.
- If data collisions are tolerated, [Space-Based Architecture](#) promises low latency and dynamic scalability.
- If a part of the system's data becomes coupled, only that part can be moved to a [Shared Repository](#), causing each instance to manage two data stores: [private and shared](#).

- Another option is to split out a [service](#) to own the coupled data and always deploy it as a single instance. The remaining parts of the system become coupled to that service, not each other.

Move all the data to a Shared Repository



Patterns: [Pool \(Shards\)](#), [Shared Database \(Shared Repository\)](#), [Load Balancer \(Proxy\)](#), [Layers](#).

Goal: don't struggle against the coupling of the shards, keep it simple and stupid.

Prerequisite: the system is not under pressure for data size or latency (which can be addressed by the further evolutions).

In case a shard needs to access data owned by any other shard, the prerequisite of the independence of shards starts to fall apart. Grab all the data of all the shards and push it into a *Shared Database*, if you can (there may be too much data or the database access may be too slow). As all the shards become identical, you'll likely add a *Load Balancer*.

Pros:

- You can choose one of the many specialized databases available.
- The stateless instances of the main application become dynamically scalable.
- Failure of a single instance affects few users.
- [Canary Release](#) is supported.

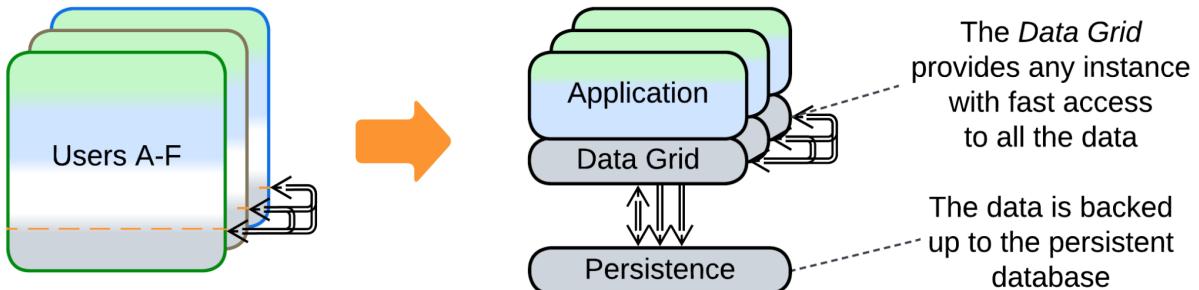
Cons:

- The database limits the system's scalability and performance.
- The *Load Balancer* and *Shared Database* increase latency and are single points of failure.

Further steps:

- [Hexagonal Architecture](#) will let you change your database in the future.
- [Space-Based Architecture](#) decreases latency by co-locating subsets of the data and your application.
- [Polyglot Persistence](#) uses multiple specialized databases, often by separating commands and queries. That may greatly relieve the primary (write) database.
- [CQRS](#) goes even further by processing read and write requests with dedicated services.

Use Space-Based Architecture



Patterns: [Space-Based Architecture \(Mesh, Shared Repository\)](#), [Shards, Layers](#).

Goal: don't struggle against the coupling between the shards, maintain high performance.

Prerequisite: data collisions are acceptable.

Space-Based Architecture is a *Mesh* of nodes which consist of the application and a cached subset of the system's data. A node broadcasts any changes to its data to other nodes and it may request any data that it needs from the other nodes. Collectively, the nodes of the *Mesh* keep the whole data cached in memory.

Though *Space-Based Architecture* may provide multiple modes of action, including [single write / multiple read](#) replicas, it is most efficient when there is no write synchronization between its nodes, meaning that data consistency is sacrificed for performance and scalability.

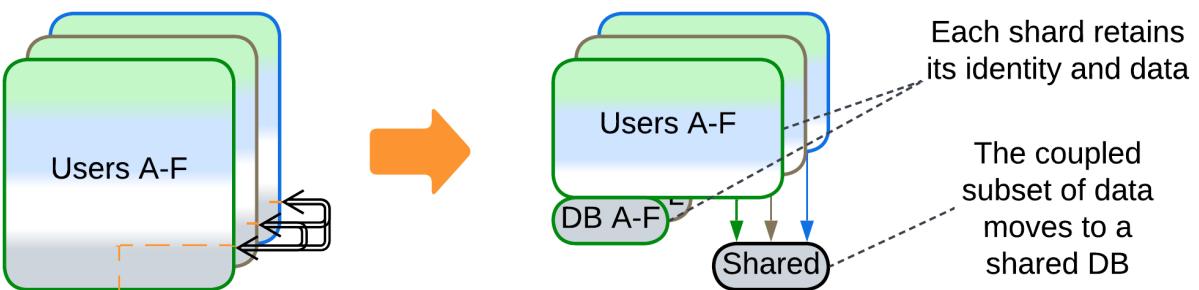
Pros:

- Unlimited dynamic scalability.
- Off-the-shelf solutions are available.
- Failure of a single instance affects few users.

Cons:

- Choose one: data collisions or mediocre performance.
- Low latency is supported only for datasets that fit in memory of a single node.
- High operational cost because the nodes exchange huge amounts of data.
- No support for analytical queries.

Use a Shared Repository for a coupled subset of the data



Patterns: [Shards, Private and Shared Databases \(Polyglot Persistence\)](#), [Shared Database \(Shared Repository\)](#), [Layers](#).

Goal: solve the coupling between shards without losing performance.

Prerequisite: the shards are coupled through a small subset of data.

If a subset of the data is accessed by all the shards, that subset can be moved to a dedicated database, which is likely to be fast if only because it is small. Using a distributed database that keeps its data synchronized on all the shards may be even faster.

This approach resembles [Shared Kernel \[DDD\]](#).

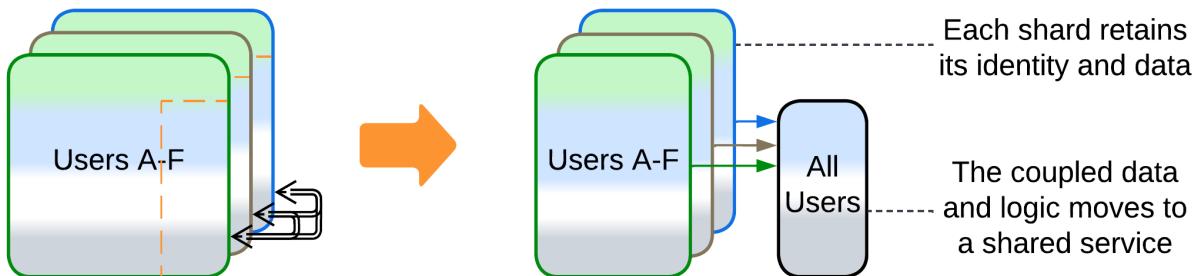
Pros:

- You can choose one of the many specialized databases available.

Cons:

- The *Shared Database* increases latency and is the single point of failure.

Split a service with the coupled data



Patterns: [Services](#), [Shards](#).

Goal: solve the coupling between the shards in an honorable way.

Prerequisite: the part of the domain which causes coupling between the shards is weakly coupled to the remaining domain.

If a part of the domain is too cohesive to be sharded, we can often move it from the main application into a dedicated service. That way the main application remains sharded while the new service exists as a single instance. In rare cases there is a chance to re-shard the new service with a sharding key which is different from the one used for sharding the main application.

This approach resembles [Shared Kernel \[DDD\]](#).

Pros:

- The main code should become a little bit simpler.
- The new service can be given to a new team.
- The new service may choose a database that best fits its needs.

Cons:

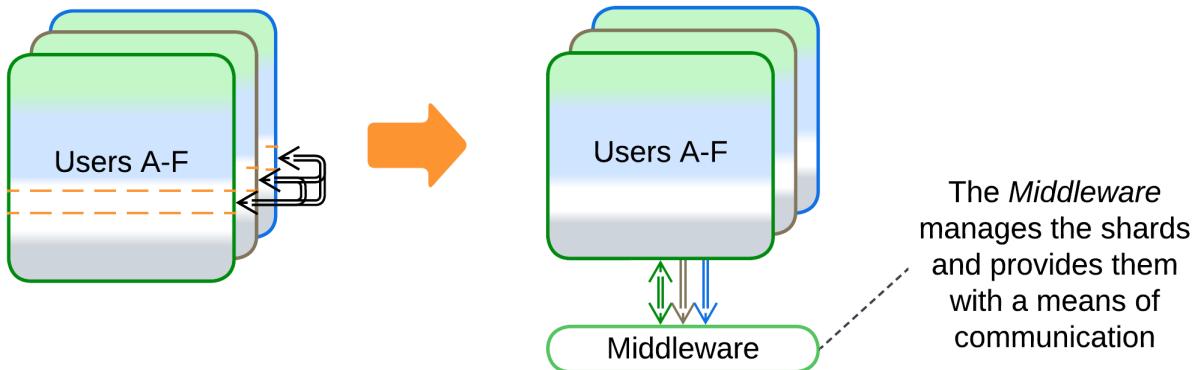
- Now it's hard to share data between the new service and the main application.
- Scenarios that use the new service are harder to debug.
- There is a moderate performance penalty for using the extra service.

Shards: share logic

Other cases are better solved by extracting the logic that manipulates multiple [shards](#):

- Splitting a [service](#) (as discussed [above](#)) yields a component that represents both shared data and shared logic.
- Adding a [Middleware](#) lets the shards communicate with each other without keeping direct connections. It also may do housekeeping: error recovery, replication, and scaling.
- A [Sharding Proxy](#) hides the existence of the shards from clients.
- An [Orchestrator](#) calls (or messages) multiple shards to serve a user request. That relieves the shards of the need to coordinate their states and actions by themselves.

Add a Middleware



Patterns: [Shards](#), [Middleware](#), [Layers](#).

Goal: simplify communication between shards, their deployment, and recovery.

Prerequisite: many shards need to exchange information, some may fail.

A *Middleware* transports messages between shards, checks their health and recovers ones which have crashed. It may manage data replication and deployment of software updates as well.

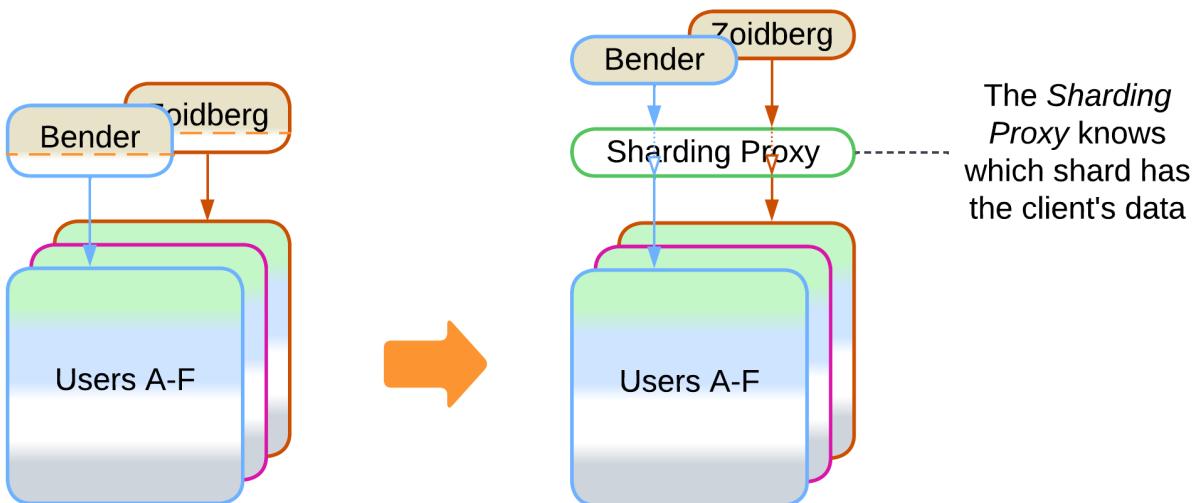
Pros:

- The shards become simpler because they don't need to track each other.
- There are many good third-party implementations.

Cons:

- Performance may degrade.
- Components of the *Middleware* are new points of failure.

Add a Sharding Proxy



Patterns: [Shards](#), [Sharding Proxy \(Proxy\)](#), [Layers](#).

Goal: simplify the code on the client side, hide your implementation from clients.

Prerequisite: each client connects directly to the shard which owns their data.

The client application may know the address of the shard which serves it and connect to it without intermediaries. That is the fastest means of communication, but it prevents you from changing the number of shards or other details of your implementation without updating all the clients, which may be unachievable. An intermediary may help.

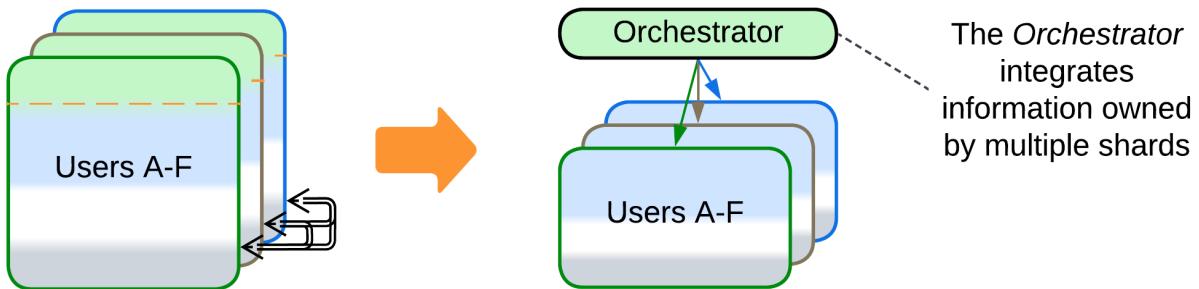
Pros:

- Your system becomes isolated from its clients.
- You can put generic aspects into the *Proxy* instead of implementing them in the shards.
- *Proxies* are readily available.

Cons:

- The extra network hop increases latency unless you deploy the *Sharding Proxy* as an [Ambassador \[DDS\]](#) co-located with every client, which brings back the issue of client software updates.
- The *Sharding Proxy* is a single point of failure unless [replicated](#).

Move the integration logic into an Orchestrator



Patterns: [Shards](#), [Orchestrator](#), [Layers](#).

Goal: isolate the shards from awareness of each other.

Prerequisite: the shards are coupled via their high-level logic.

When a high-level scenario uses multiple shards ([Scatter-Gather](#) and [MapReduce](#) are the simplest examples), the way to follow is to extract all such scenarios into a dedicated stateless module. That makes the shards independent of each other.

Pros:

- The shards don't have to be aware of each other.
- The high-level logic can be written in a high-level language by a dedicated team.
- The high-level logic can be deployed independently.
- The main code should become much simpler.

Cons:

- Latency will increase.
- The *Orchestrator* becomes a single point of failure with a good chance to corrupt your data.

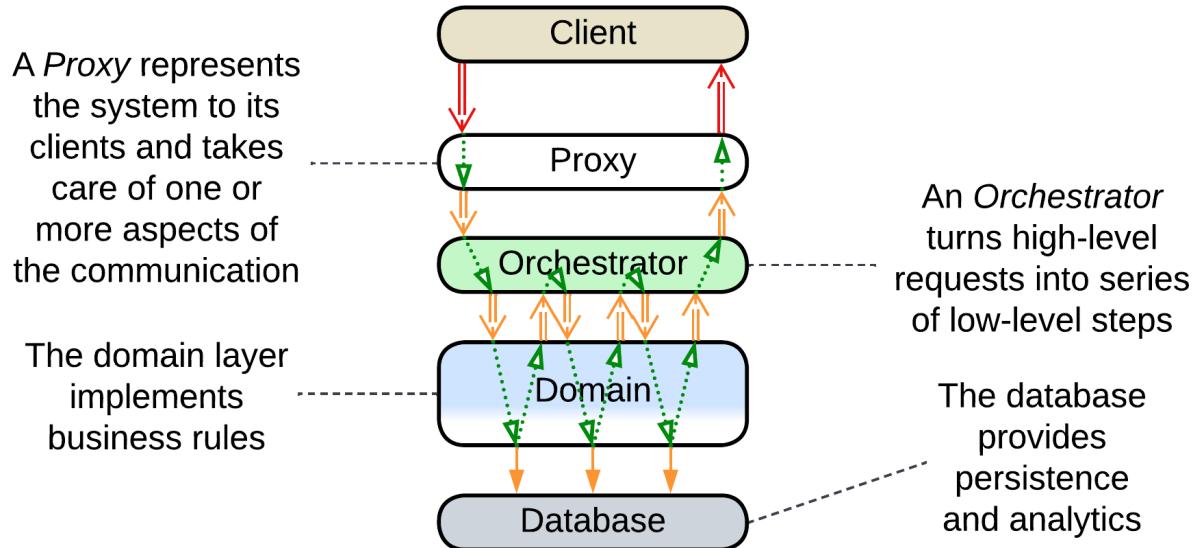
Further steps:

- [Shard or replicate the Orchestrator](#) to support higher load and to remain online if it fails.
- *Persist* the *Orchestrator* (give it a dedicated database) to make sure that it does not leave half-committed transactions upon failure.
- *Divide* the *Orchestrator* [into Backends for Frontends](#) or a [SOA-style layer](#) if you have multiple kinds of clients or workflows, correspondingly.

Layers: make more layers

Not all the layered architectures are equally layered. A [Monolith](#) with a [Proxy](#) or database has already stepped into the realm of [Layers](#) but is far from reaping all of its benefits. It may continue its journey in a few ways that [were earlier discussed](#) for *Monolith*:

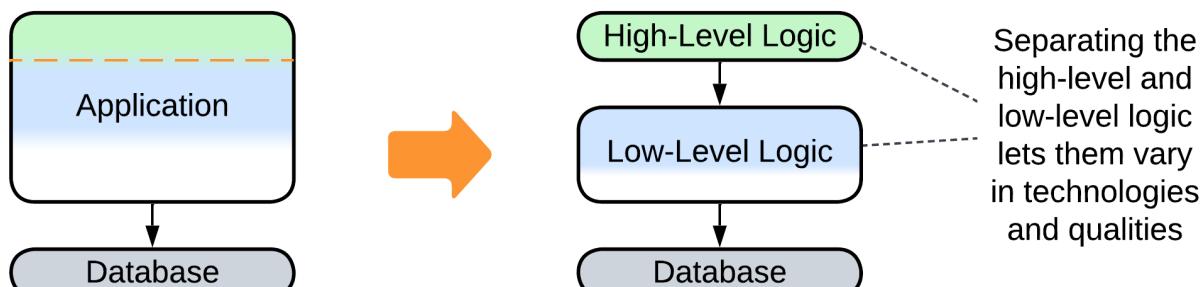
- Employing a *database* (if you don't use one) lets you rely on a thoroughly optimized state-of-the-art subsystem for data processing and storage.
- *Proxies* are similarly reusable generic modules to be added at will.
- Implementing an *Orchestrator* on top of your system may improve programming experience and runtime performance for your clients.



It is also common to:

- Segregate the business logic into two *layers*.

Split the business logic into two layers



Patterns: [Layers](#).

Goal: let parts of the business logic vary in qualities, improve the structure of the code.

Prerequisite: the high-level and low-level logic are loosely coupled.

It is often possible to split a backend into integration ([orchestration](#)) and domain layers. That allows for one team to specialize in customer use cases while the other one delves deep into the domain knowledge and infrastructure.

Pros:

- You get an extra development team.
- High-level use cases may be deployed separately from business rules.
- The layers may diverge in technologies and styles.
- The code may [become less complex](#).

Cons:

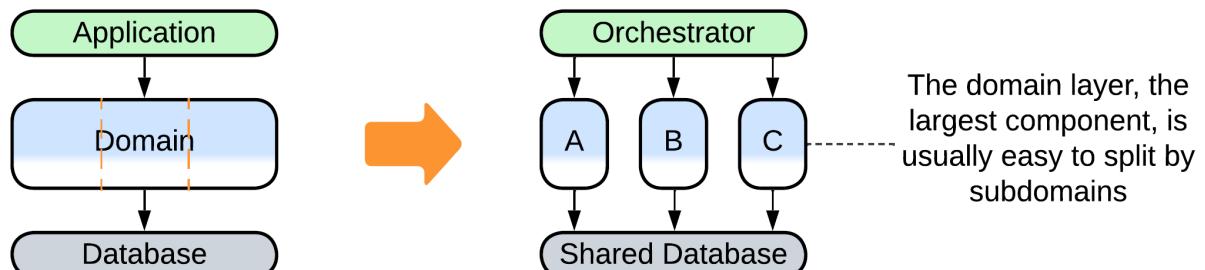
- There is a small performance penalty.
- In-depth debugging becomes harder.

Layers: help large projects

The main drawback (and benefit) of [Layers](#) is that much or all of the business logic is kept together in one or two components. That allows for easy debugging and fast development in the initial stages of the project but slows down and complicates work as the project [grows in size](#). The only way for a growing project to survive and continue evolving at a reasonable speed is to divide its business logic into several smaller, thus less [complex](#), components that match subdomains (*bounded contexts* [[DDD](#)]). There are several options for such a change whose applicability depends on the domain:

- The middle layer with the main business logic can be divided into [Services](#) leaving the upper [Orchestrator](#) and lower [database](#) layers intact for future evolutions.
- Sometimes the business logic can be represented as a set of directed graphs which is known as [Event-Driven Architecture](#).
- If you are lucky, your domain is naturally a [Top-Down Hierarchy](#).

Divide the domain layer into Services



Patterns: [Services](#), [Shared Database \(Shared Repository\)](#), [Orchestrator](#).

Goal: make the code simpler and let several teams work on the project efficiently.

Prerequisite: the low-level business logic comprises loosely coupled subdomains.

It is very common for a system's domain to consist of weakly interacting *bounded contexts* [[DDD](#)]. They are integrated through high-level use cases and/or [relations in data](#). For such a system it is relatively easy to divide the domain logic into Services while leaving the integration and data layers shared.

Pros:

- You get multiple specialized development teams.
- The largest and most complex piece of code is split into several smaller components.
- There is more flexibility with deployment and scaling.

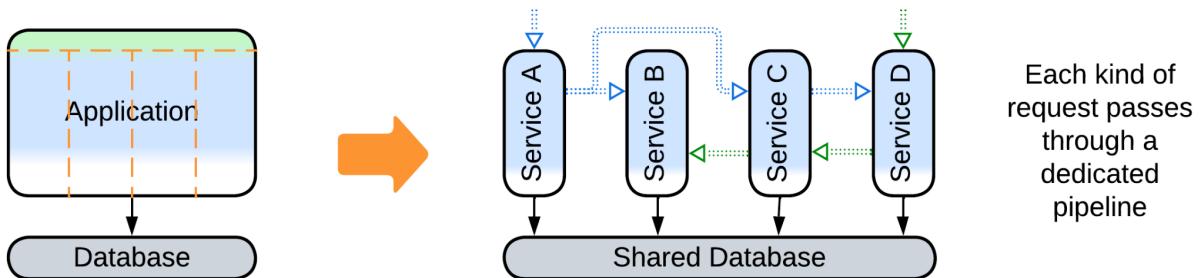
Cons:

- Future changes in the overall structure of the domain will be harder to implement.
- System-wide use cases become somewhat harder to debug as they span over many components.
- Performance may degrade if the *Services* and *Orchestrator* become distributed.

Further steps:

- Continue by splitting the *Orchestrator* and *database*, resulting in [Orchestrated Three-Layered Services](#).
- Divide the *Orchestrator* (by type of client) into [Backends for Frontends](#).
- Use multiple databases in [Polyglot Persistence](#).
- Scale well with [Space-Based Architecture](#).

Build an Event-Driven Architecture over a Shared Database



Patterns: [Event-Driven Architecture \(Pipeline \(Services\)\)](#), [Shared Database \(Shared Repository\)](#).

Goal: untangle the code, support multiple teams, improve scalability.

Prerequisite: use cases are sequences of loosely coupled coarse-grained steps.

If your system has a well-defined workflow for processing every kind of input request, it can be divided into several [subdomain services](#), each hosting a few related steps of multiple use cases. Each service subscribes to inputs from other services and/or system's clients and publishes output events.

Pros:

- The code is divided into much smaller (and simpler) segments.
- It is easy to add new steps or use cases as the structure is quite flexible.
- You open a way to having several almost independent teams, one per service.
- You can achieve flexible deployment and scaling as the services are stateless, but you need a *Middleware* for that.
- The architecture naturally supports event replay as the means of reproducing bugs or testing / benchmarking individual components.
- There is no need for explicit scheduling or thread synchronization.

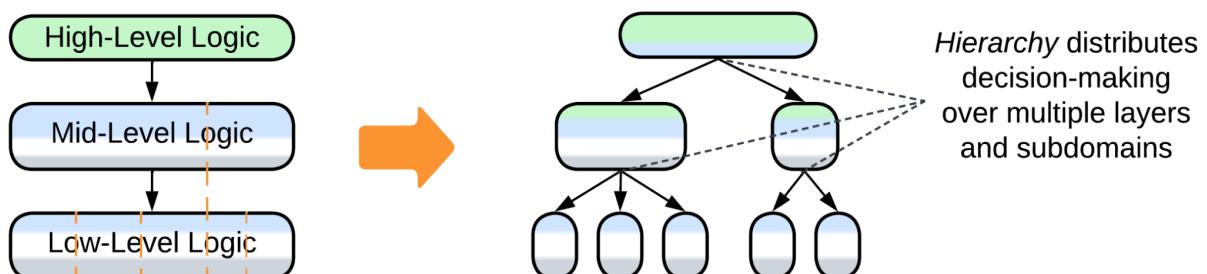
Cons:

- The system as a whole is hard to debug.
- You will have to live with high latency.
- You may end up with too many components which are interconnected in too many ways.

Further steps:

- Add a [Middleware](#) that supports scaling and failure recovery.
- Split the [Shared Database](#) by subdomain, yielding [Choreographed Two-Layered Services](#).
- Scale with [Space-Based Architecture](#).
- Extract the logic of use cases into an [Orchestrator](#).

Build a Top-Down Hierarchy



Patterns: [Top-Down Hierarchy \(Hierarchy\)](#).

Goal: untangle the code, support multiple teams, earn fine-grained scalability.

Prerequisite: the domain is hierarchical.

Splitting the lower layers into independent components with identical interfaces simplifies the managing code and allows the managed components to be deployed, developed, and run independently of each other. Ideally, the mid-layer components should participate in decision-making so that the uppermost component is kept relatively simple.

Pros:

- Hierarchy is easy to develop and support with multiple teams.
- Individual components are straightforward to modify or replace.
- The components scale, deploy, and run independently.
- The system is quite fault tolerant.

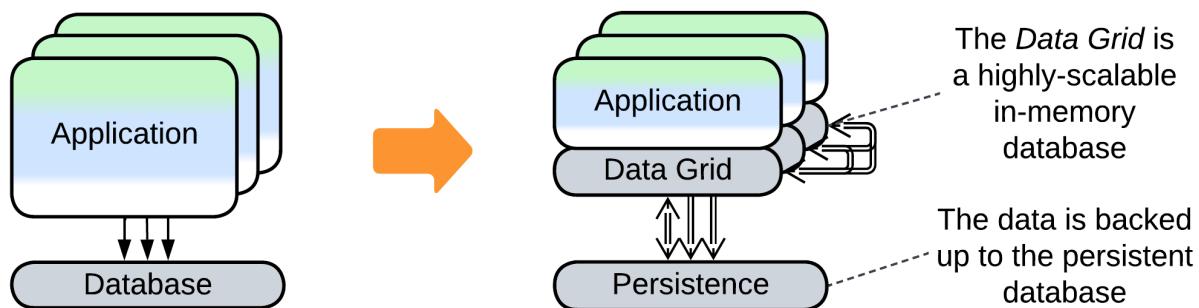
Cons:

- It takes time and skill to figure out good interfaces.
- There are many components to administer.
- Latency is suboptimal for system-wide use cases.

Layers: improve performance

There are several ways to improve the performance of a [layered system](#). One we have [already discussed](#) for [Shards](#):

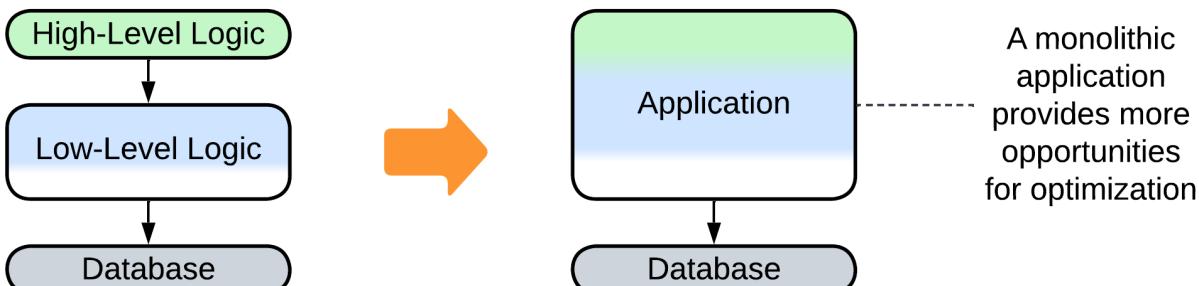
- [Space-Based Architecture](#) co-locates the database and business logic and scales both dynamically.



Others are new here and thus deserve more attention:

- Merging several layers improves latency by eliminating the communication overhead.
- Scaling some of the layers may improve throughput but degrade latency.
- [Polyglot Persistence](#) is the name for using multiple specialized databases.

Merge several layers



Patterns: [Layers](#) or [Monolith](#)

Goal: improve performance.

Prerequisite: the layers share programming language, hardware setup and qualities.

If your system's development [is finished](#) (no changes are expected) and you really need that extra 5% performance improvement, then you can try merging everything back into a *Monolith* or a [3-Tier](#) system (front, back, data).

Pros:

- Enables aggressive performance optimizations.
- The system may become easier to debug.

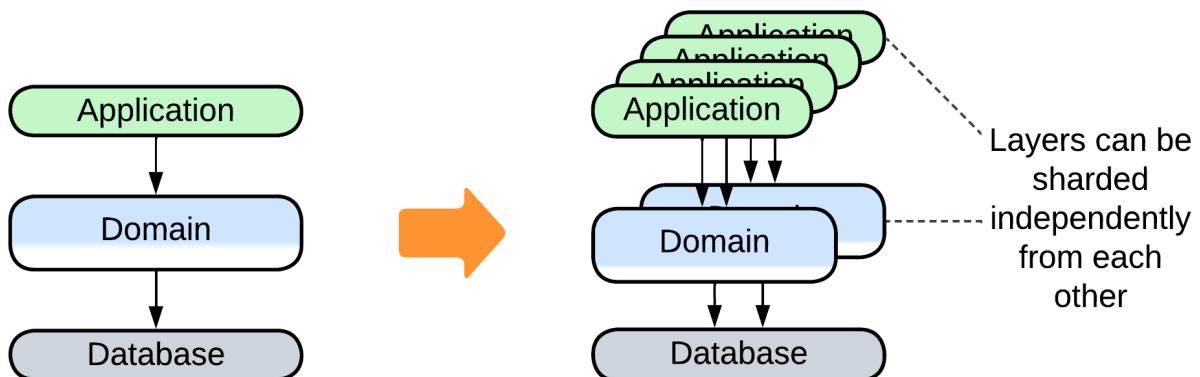
Cons:

- The code is frozen – it will be much harder to evolve.
- Your teams lose the ability to work independently.

Further steps:

- [Shard](#) the entire system.

Scale individual layers



Patterns: [Layers](#), [Shards](#), often [Load Balancer \(Proxy\)](#).

Goal: scale the system.

Prerequisite: some layers are [stateless](#) or limited to the [data of a single client](#).

Multiple instances or layers can be created, with their number and deployment [varying from layer to layer](#). That may work seamlessly if each instance of the layer which receives an event which can start a use case knows the instance of the next layer to communicate to. Otherwise you will need a *Load Balancer*.

Pros:

- Flexible scalability.
- Better fault tolerance.
- Co-deployment with clients is possible.

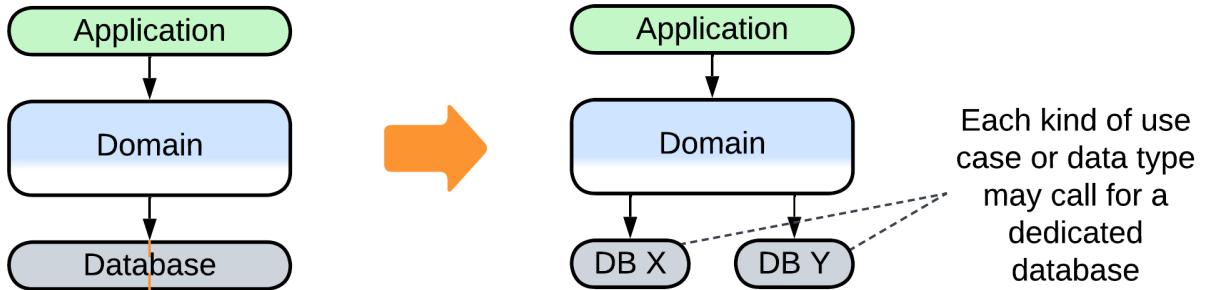
Cons:

- More complex operations (more parts to keep an eye on).

Further steps:

- [Space-Based Architecture](#) scales the data layer.
- [Polyglot Persistence](#) improves performance of the data layer.

Use multiple databases



Each kind of use case or data type may call for a dedicated database

Patterns: [Layers](#), [Polyglot Persistence](#).

Goal: optimize performance of data processing.

Prerequisite: there are isolated use cases for or subsets of the data.

If you have separated *commands* (write requests) from *queries* (read requests), you can serve the queries with [read-only replicas](#) of the database while the main database is reserved for the commands.

If your types of data or data processing algorithms vary, you may deploy several [specialized databases](#), each matching a subset of your needs. That lets you achieve the best performance in widely diverging cases.

Pros:

- The best performance for all the use cases.
- Specialized data processing algorithms out of the box.
- Replication may help with error recovery.

Cons:

- Someone will need to learn and administer all those databases.
- Keeping the databases consistent takes effort and the replication delay may negatively affect UX.

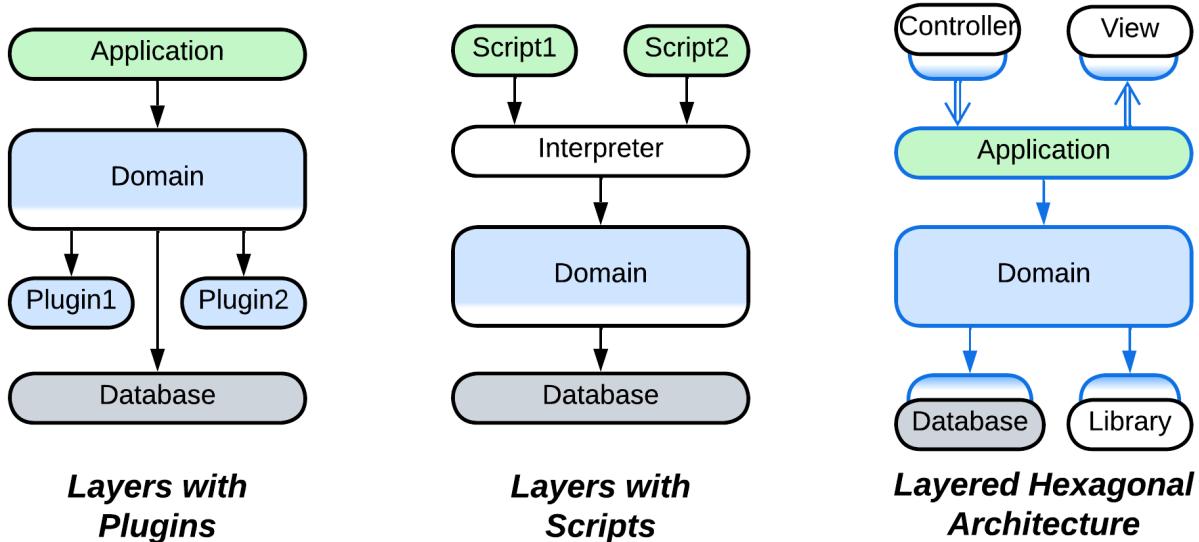
Further steps:

- Serve read and write requests with different backends according to [Command-Query Responsibility Segregation \(CQRS\)](#).
- Separate the backend into [services](#) which match the already separated databases.

Layers: gain flexibility

The last group of evolutions to consider is about making the system more adaptable. We have [already discussed](#) the following evolutions for [Monolith](#):

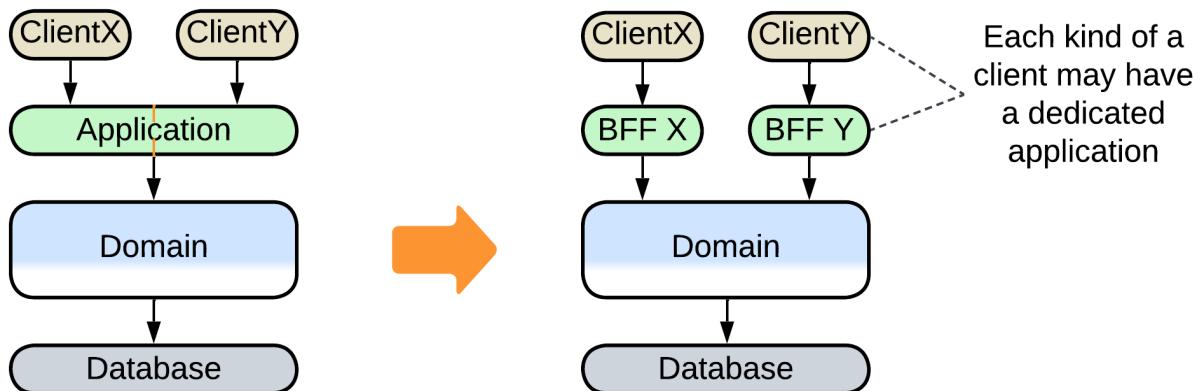
- The behavior of the system may be modified through [Plugins](#).
- [Hexagonal Architecture](#) protects the business logic from dependencies on libraries and databases.
- [Scripts](#) allow for customization of the system's logic on a per client basis.



There is also a new evolution that modifies the upper (orchestration) layer:

- The *orchestration layer* may be split into [Backends for Frontends](#) to match the needs of several kinds of clients.

Divide the orchestration layer into Backends for Frontends



Patterns: [Layers](#), [Backends for Frontends aka BFFs](#).

Goal: let each kind of client get a dedicated development team.

Prerequisite: no high-level logic is shared between client types.

It is possible that your system has different kinds of users, e.g. buyers, sellers, and admins; or web and mobile applications. It may be easier to support a separate integration module per kind of client than to keep all the unrelated code together in a single integration layer.

Pros:

- Each kind of client gets a dedicated team which may choose best fitting technologies.
- You get rid of the single large codebase of the integration layer.

Cons:

- There is no good way to share code between the *BFFs* (in [naive implementation](#)).
- There are new components to administer.

Further steps:

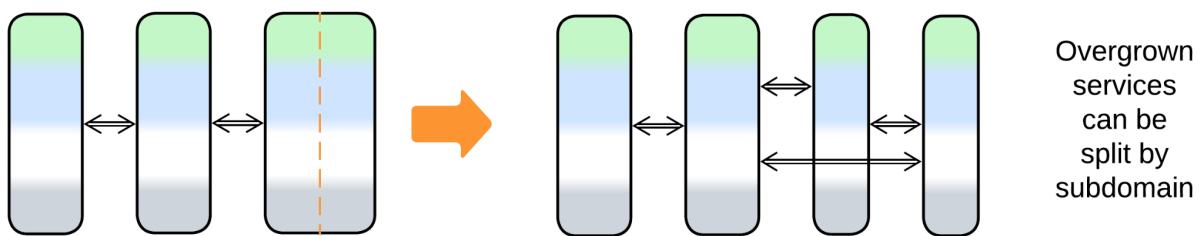
- [Evolve the BFFs](#) through adding a shared *layer* or [Sidecars](#) for common functionality.

Services: add or remove services

[Services](#) work well when each service matches a subdomain and is developed by a single team. If those premises change, you'll need to restructure the system:

- A new feature request may emerge outside of any of the existing subdomains, creating a new service.
- A service may grow too large to be developed by a single team, calling for division.
- Two services may become so strongly coupled that they fare better when merged together.
- The entire system may need to be glued back into a [Monolith](#) if domain knowledge changes or interservice communication strongly degrades performance.

Add or split a service



Patterns: [Services](#).

Goal: get one more team to work on the project, decrease the size of an existing service.

Prerequisite: there is a loosely coupled (new or existing) subdomain that does not have a dedicated service (yet).

If you need to add a new functionality that does not naturally fit into one of the existing services, you may create a new service and maybe get a new team for it.

If one of your services has grown too large, you should look for a way to subdivide it (likely through a [Cell](#) stage with a shared [Orchestrator](#) and [database](#)) to decrease the size and, correspondingly, complexity of its code and get multiple teams to work on the resulting (sub)services. However, that makes sense only if the old service is not highly cohesive – otherwise [the resulting subsystem may be more complex](#) than the original service.

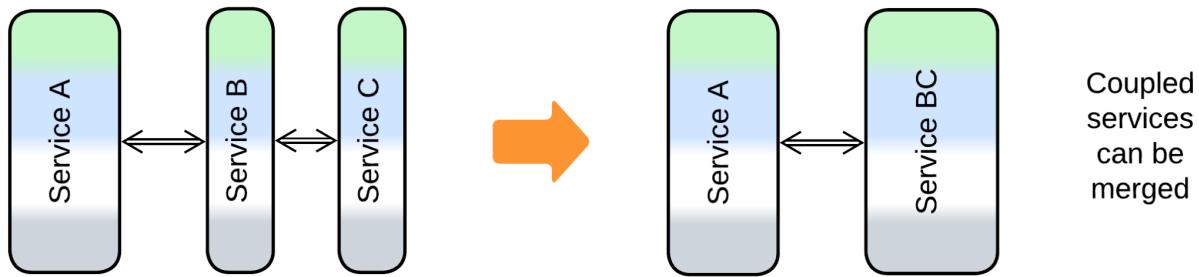
Pros:

- You get an extra development team.
- The complexity of the code decreases (splitting an existing service) or does not increase (adding a new one).
- The new service is independently scalable.

Cons:

- You add to the operations complexity by creating a new system component and several inter-component dependencies.
- There is a new point of failure, which means that bugs and outages become more likely.
- Performance (or at least latency and cost efficiency) of the system will deteriorate because interservice communication is slow.
- You may have a hard time debugging use cases that involve both the old and new service.

Merge services



Patterns: [Services](#), [Monolith](#) or [Layers](#).

Goal: accept the coupling of subdomains and improve performance.

Prerequisite: the services use compatible technologies.

If you see that several services communicate with each other almost as intensely as they call their internal methods, they probably belong together.

If your use cases have too high a latency or you pay too much for CPU and traffic, the issue may originate with the interservice communication and merging the services should help. No services, no pain.

Alternatively, as domain knowledge changes [DDD], you may have to merge much of the code together only to subdivide it later along updated subdomain boundaries. Which means you face [lots of work for no reason](#).

Pros:

- Improved performance.
- It becomes easy for parts of the merged code to access each other and share data.
- The new merged service or *Monolith* is easier to debug than the original *Services*.

Cons:

- The development teams become even more interdependent.
- There is no good way to vary qualities by subdomain.
- You lose granular scalability by subdomain.
- The merged codebase may be too large for comfortable development.
- If anything fails, everything fails.

Services: add layers

The most common modifications to a [system of Services](#) involve supplementary system-wide *layers* which compensate for the inability of the *Services* to share anything among themselves:

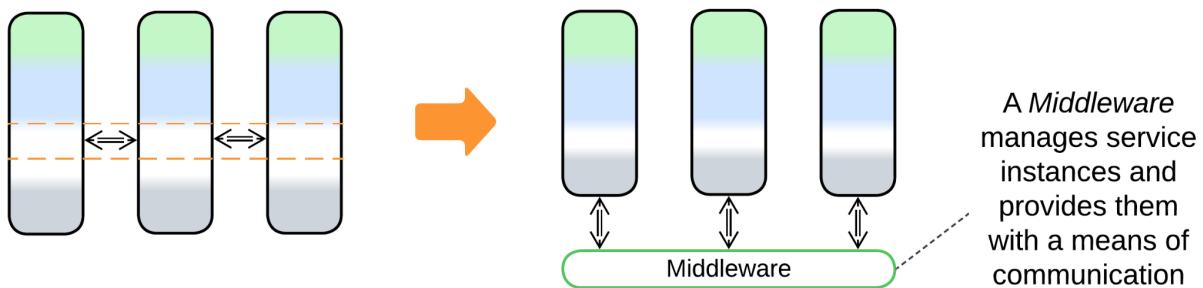
- A [Middleware](#) knows of all the deployed service [instances](#). It mediates communication between them and may manage their scaling and failure recovery.
- [Sidecars](#) [DDS] of a [Service Mesh](#) make a virtual layer of [shared libraries](#) for the [Microservices](#) it hosts.
- A [Shared Database](#) simplifies the initial phases of development and provides data consistency and [interservice communication](#).
- [Proxies](#) stand between the system and its clients and take care of shared aspects that otherwise would need to be implemented by every service.
- An [Orchestrator](#) is the single place for the high-level logic of every use case.

Those layers may also be merged into [Combined Components](#):

- [Message Bus](#) is a [Middleware](#) that supports multiple protocols.

- [API Gateway](#) combines [Gateway](#) (a kind of [Proxy](#)) and [Orchestrator](#).
- [Event Mediator](#) is an [orchestrating Middleware](#).
- [Shared Event Store](#) combines [Middleware](#) and [Shared Repository](#).
- [Enterprise Service Bus \(ESB\)](#) is an [orchestrating Message Bus](#).
- [Space-Based Architecture](#) employs all the four layers: [Gateway](#), [Orchestrator](#), [Shared Repository](#) and [Middleware](#).

Add a Middleware



Patterns: [Middleware](#), [Services](#).

Goal: take care of scaling, recovery, and interservice communication without programming it.

Prerequisite: communication between the services is uniform.

Distributed systems may fail in a zillion ways. You want to ruminate neither on that nor on [heisenbugs](#). And you probably want to have a framework for scaling the services and restarting them after failure. Get a third-party *Middleware*! Let your programmers write the business logic, not infrastructure.

Pros:

- You don't invest your time in infrastructure.
- Scaling and error recovery are made easy.

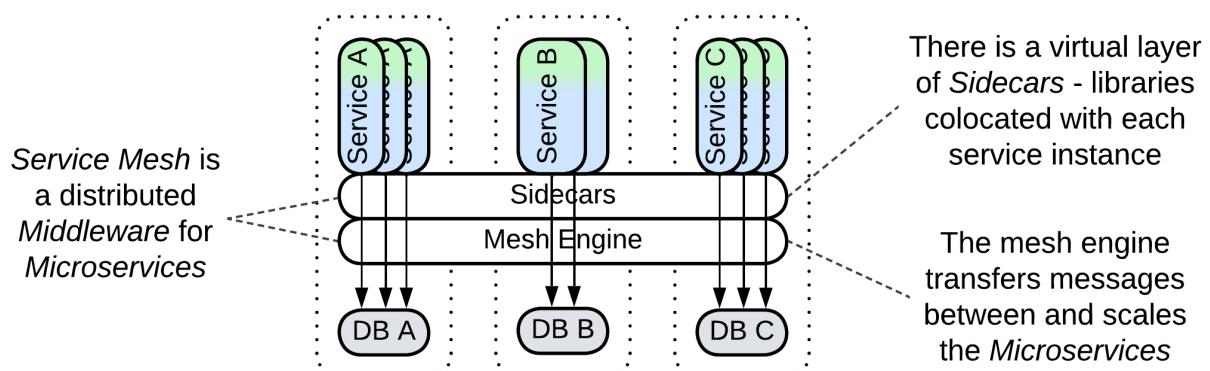
Cons:

- There may be a performance penalty which becomes worse for uncommon patterns of communication.
- The *Middleware* may be a single point of failure.

Further steps:

- Use a [Service Mesh](#) for dynamic scaling and as a way to implement [shared aspects](#).

Use a Service Mesh



Patterns: [Service Mesh \(Mesh, Middleware\)](#), [Sidecar \(Proxy\)](#), [Services](#).

Goal: support dynamic scaling and interservice communication out of the box; share libraries among the services.

Prerequisite: service instances are mostly stateless.

The Microservices architecture boasts dynamic scaling under load thanks to its *Mesh-based Middleware*. It also allows for the services to share libraries in *Sidecars* [DDS] – additional containers co-located with each service instance – to avoid duplication of generic code among the services.

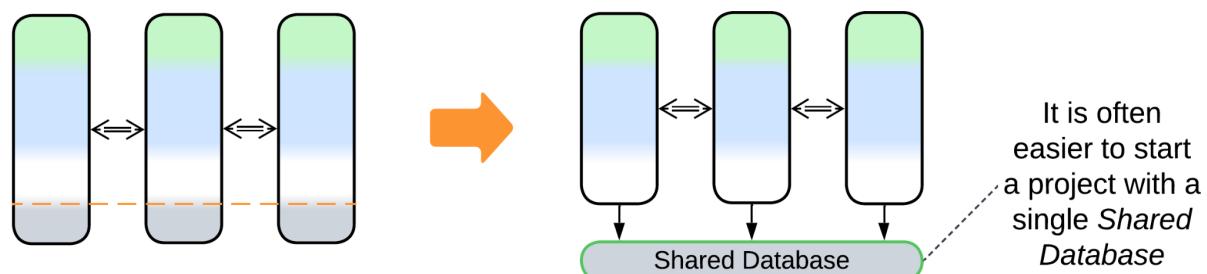
Pros:

- Dynamic scaling and error recovery.
- Available out of the box.
- Provides a way to implement shared aspects (cross-cutting concerns) once and use the resulting libraries in every service.

Cons:

- Performance degrades because of the complex distributed infrastructure.
- You may suffer vendor lock-in.

Use a Shared Repository



Patterns: Shared Repository, Services.

Goal: let the services share data, don't invest in operating multiple databases.

Prerequisite: the services use a uniform approach to persisting their data.

You don't really need every service to have a private database. A shared one is enough in many cases.

Pros:

- It is easy for the services to share and synchronize data.
- Lower operational complexity.

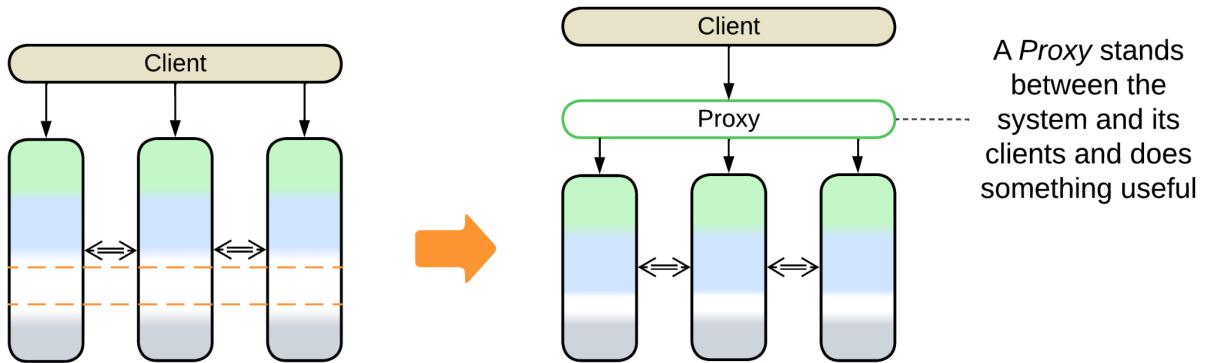
Cons:

- All the services depend on the database schema which becomes hard to alter.
- The single database will limit performance of the system.
- It may also become a single point of failure.

Further steps:

- Space-Based Architecture scales the data layer but it is a simple key-value store.
- Polyglot Persistence is about having multiple specialized databases.

Add a Proxy



Patterns: [Proxy](#), [Services](#).

Goal: use a common infrastructure component on behalf of your entire system.

Prerequisite: the system serves its clients in a uniform way.

Putting a generic component between the system and its clients helps the programmers concentrate on business logic rather than protocols, infrastructure or even security.

Pros:

- You get a choice of generic functionality without investing development time.
- It is an additional layer that isolates your system from both clients and attackers.

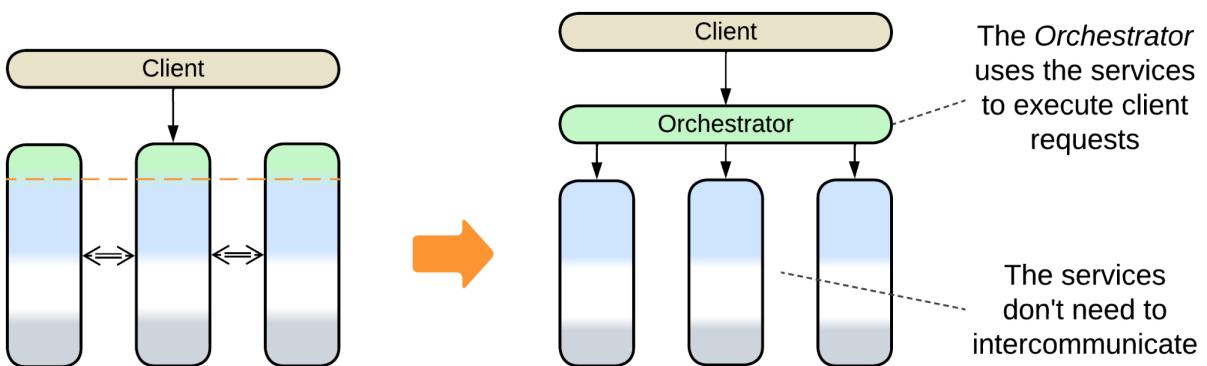
Cons:

- There is a latency penalty caused by the extra network hop.
- Each *Proxy* may be a single point of failure or at least needs some admin oversight.

Further steps:

- You can always add another kind of [Proxy](#).
- If there are multiple clients that differ in their protocols, you can employ a stack of *Proxies* per client, resulting in [Backends for Frontends](#).

Use an Orchestrator



Patterns: [Orchestrator](#), [Services](#).

Goal: have the high-level logic of use cases distilled as intelligible code.

Prerequisite: the use cases comprise sequences of high-level steps (which is very likely to be true for a system of [subdomain services](#)).

When a use case jumps over several services in a dance of [choreography](#), there is no easy way to understand it as there is no single place to see it in the code. It may be even worse with [Pipelined](#) systems where use cases are embodied in the structure of event channels between the components.

Extract the high-level business logic from the choreographed services or their interconnections and put it into a dedicated component.

Pros:

- You are not limited in the number and complexity of use cases anymore.
- Global use cases become much easier to debug.
- You have a new team dedicated to the interaction with customers, freeing the other teams to study their parts of the domain or work on improvements.
- Many changes in the high-level logic can be implemented and deployed without touching the main services.
- The extra layer decouples the main services from the system's clients and from each other.

Cons:

- There is a performance penalty because the number of messages per use case doubles.
- The *Orchestrator* may become a single point of failure.
- Some flexibility is lost as the *Orchestrator* couples qualities of the services.

Further steps:

- If there are several clients that strongly vary in workflows, you can apply [Backends for Frontends](#) with an *Orchestrator* per client.
- If the *Orchestrator* grows too large, it [can be divided](#) into layers, services or both, the latter option resulting in a [Top-Down Hierarchy](#).
- The *Orchestrator* can be [scaled](#) and can have its own database.

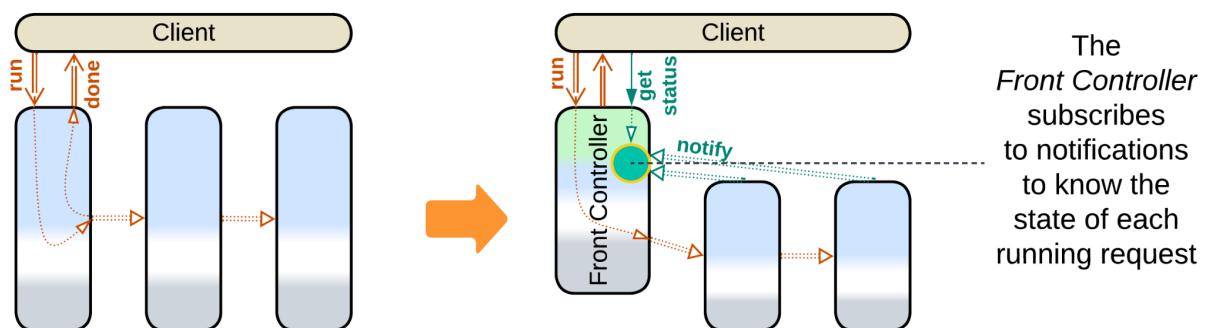
Pipeline:

[Pipeline](#) inherits its set of evolutions from [Services](#). Components can be added, split in two, merged or replaced. Many systems employ a [Middleware](#) (pub/sub or pipeline framework), [Shared Repository](#) (which may be a database or file system) or [Proxies](#).

There are a couple of *Pipeline*-specific evolutions:

- The first service of the *Pipeline* can be promoted to [Front Controller](#) [SAHP] which tracks status updates for every request it handles.
- Adding an *Orchestrator* turns a *Pipeline* into [Services](#). As the high-level business logic moves to the orchestration layer, the services don't need to interact directly, the interservice communication channels disappear and the system becomes identical to [Orchestrated Services](#).

Promote a service to Front Controller



Patterns: [Front Controller](#) ([Polyglot Persistence](#), [Orchestrator](#)), [Pipeline](#) ([Services](#)).

Goal: allow for clients to query the state of their requests.

Prerequisite: request processing steps are slow (may depend on human action).

If request processing steps require heavy calculations or manual action, clients may want to query the status of their requests and analysts may want to see bottlenecks in the *Pipeline*. Let the first service in the *Pipeline* track the state of all the running requests by subscribing to status notifications from other services.

Pros:

- The state of each running request is readily available.

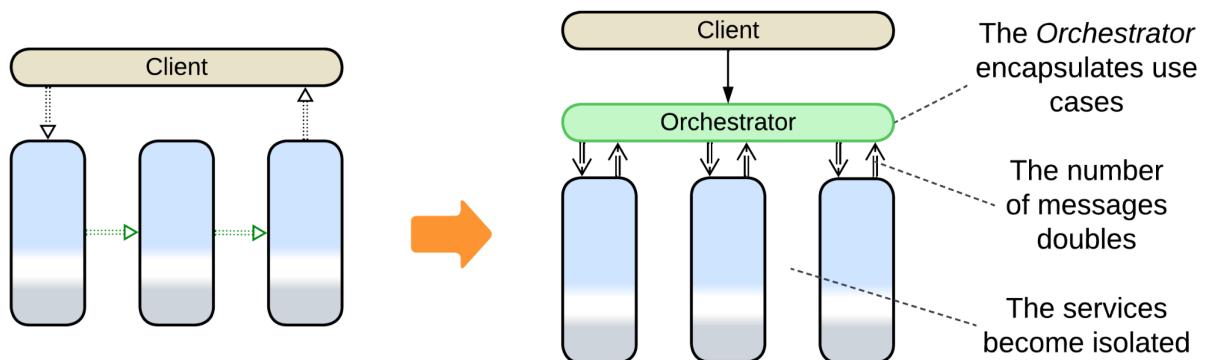
Cons:

- The first service in the pipeline depends on every other service.

Further steps:

- The *Front Controller* may be further promoted to *Orchestrator* if there is a need to support many complex scenarios.

Add an Orchestrator



Patterns: [Orchestrator](#), [Services](#).

Goal: support many use cases.

Prerequisite: performance degradation is acceptable.

When a *choreographed* system is extended with more and more use cases, it is very likely to fall into integration hell where nobody understands how its components interrelate. Extract the workflow logic into a dedicated service.

Pros:

- New use cases are easy to add.
- Complex scenarios are supported.
- The services don't depend on each other.
- There is a single client-facing team, other teams are not under pressure from the business.
- It is easier to run actions in parallel.
- Global scenarios become debuggable.
- The services don't need to be redeployed when the high-level logic changes.

Cons:

- The number of messages in the system doubles, thus its performance may degrade.
- The *Orchestrator* may become a development and performance bottleneck or a single point of failure.

Further steps:

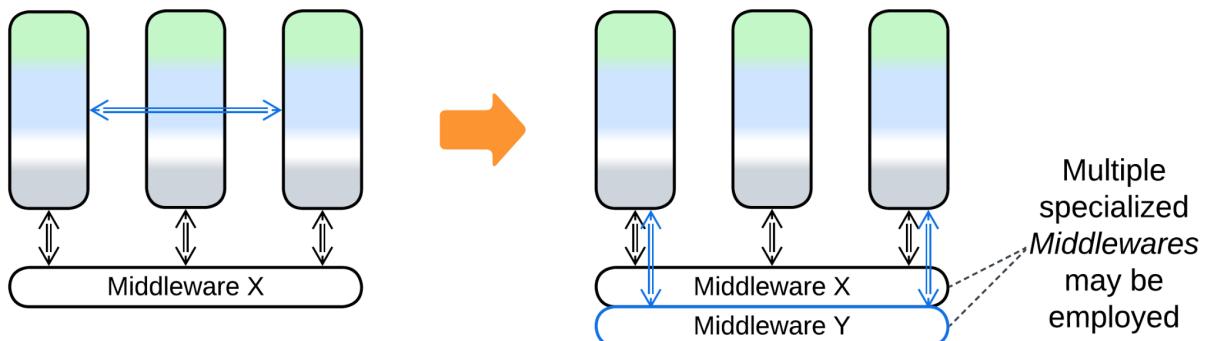
- If there are several clients that strongly vary in workflows, you can apply [Backends for Frontends](#) with an *Orchestrator* per client.
- If the *Orchestrator* grows too large, it can be [divided](#) into layers, services or both, the latter option resulting in a [Top-Down Hierarchy](#).
- The *Orchestrator* can be [scaled](#) and can have its own database.

Middleware:

A [Middleware](#) is unlikely to be removed (though it may be replaced) once it is built into a system. There are few evolutions as a *Middleware* is a third-party product and is unlikely to be messed with:

- If the *Middleware* in use does not fit the preferred mode of communication between some of your services, there is an option to deploy a second specialized *Middleware*.
- If several existing systems need to be merged, that is accomplished by adding yet another layer of *Middleware*, resulting in a [Bottom-Up Hierarchy \(Bus of Buses\)](#).

Add a secondary middleware



Patterns: [Middleware](#).

Goal: support specialized communication between [scaled](#) services.

Prerequisite: the system relies on a *Middleware* for scaling.

If the current *Middleware* is too generic for the system's needs, you can add another one for specialized communication. The new *Middleware* does not manage the instances of the services.

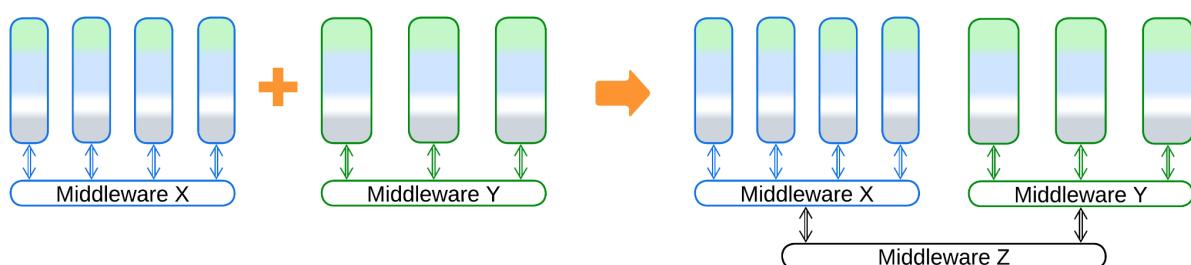
Pros:

- Supports specialized communication with no need to write code for tracking the instances of services.

Cons:

- You still need to notify the new *Middleware* when an instance of a service is created or dies.
- There is an extra component to administer.

Merge two systems by building a Bottom-Up Hierarchy



Patterns: [Bottom-up Hierarchy \(Hierarchy, Middleware\)](#).

Goal: integrate two systems without a heavy refactoring.

Prerequisite: both systems use *Middlewares*.

If we cannot change the way each subsystem's services use its *Middleware*, we should add a new *Middleware* to connect the existing *Middlewares*.

Pros:

- No need to touch anything in the existing services.

Cons:

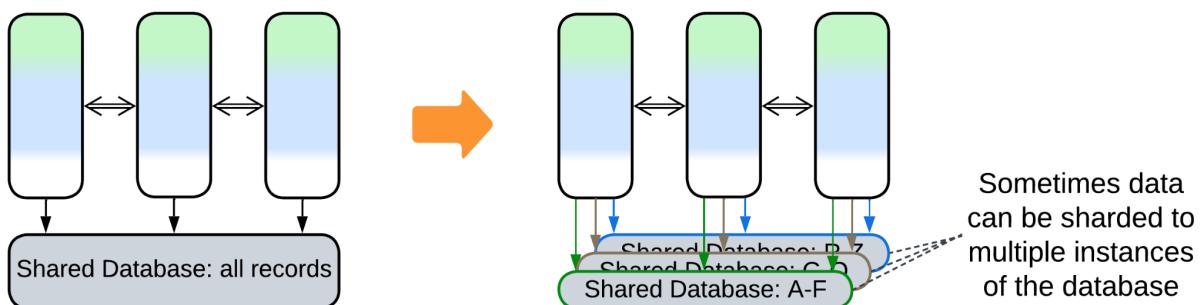
- Performance suffers from the double conversion between protocols.
- There is a new component to fail (miserably).

Shared Repository:

Once a database appears, it is unlikely to go away. I see the following evolutions to improve performance of the data layer:

- [Shard](#) the database.
- Use [Space-Based Architecture](#) for dynamic scalability.
- Divide the data into a private database per service.
- Deploy specialized databases ([Polyglot Persistence](#)).

Shard the database



Patterns: [Sharding \(Shards\)](#), [Shared Repository](#), maybe [Sharding Proxy](#).

Goal: increase capacity and performance of the database.

Prerequisite: the data is shardable (consists of independent records).

If your database is overloaded and the data which it contains describes independent entities (users, companies, sales) you can deploy multiple instances of the database with subsets of the data distributed among them. You will need to deploy a *Sharding Proxy* or the services will have to find out which database *shard* to access by themselves, likely through hashing the record's *primary key* [[DDIA](#)]. There is also a good chance that several smaller tables will have to be replicated to all the shards or moved to a [dedicated Shared Database](#) (resulting in [Polyglot Persistence](#)).

Modern distributed databases support sharding out of the box, but an overgrown table may still impact the performance of the database.

Pros:

- Unlimited scalability.
- You don't need to change your database vendor.
- Failure of a single database instance affects few users.

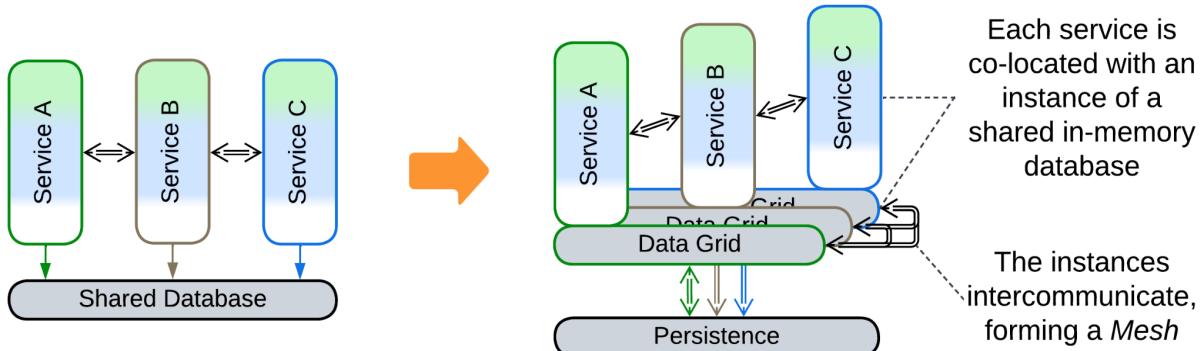
Cons:

- You need to manage many instances of the database.
- The application or a custom script may have to synchronize shared tables among the instances.
- There is no way to do joins or run aggregate functions (such as *sum* or *count*) over multiple shards – all that logic moves to the services that use the database.

Further steps:

- [Polyglot Persistence](#) or [CQRS](#) describe pre-calculating aggregates into another analytical database ([Reporting Database](#)).
- [Space-Based Architecture](#) may be cheaper as it scales dynamically. However, in its default and highly performant configuration it is prone to write collisions.

Use Space-Based Architecture



Patterns: [Space-Based Architecture \(Mesh, Shared Repository\)](#).

Goal: scale throughput of the database dynamically.

Prerequisite: data collisions are acceptable.

Space-Based Architecture (SBA) duplicates contents of a persistent database to a distributed in-memory cache co-located with the services managed by the [SBA's Middleware](#). That makes most database access operations very fast unless one needs to avoid write collisions. The *Mesh Middleware* autoscales both the services and the associated data cache under load, granting nearly perfect scalability. However, this architecture is costly because of the amount of traffic and CPU time spent on replicating data between the *Mesh nodes*.

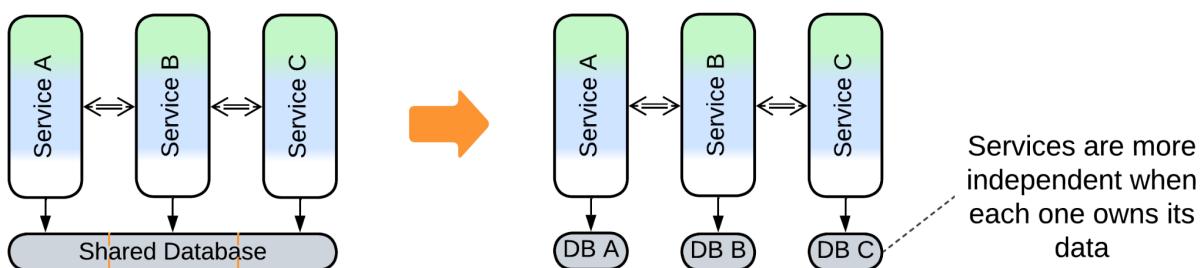
Pros:

- Nearly unlimited dynamic scalability.
- Off-the-shelf solutions are available.
- Very high fault tolerance.

Cons:

- Choose one: data collisions or poor performance.
- Low latency is guaranteed only when the entire dataset fits in the memory of a node.
- High operational cost because the nodes will send each other lots of data.
- No support for analytical queries.

Move the data to private databases of services



Patterns: [Services](#) or [Shards, Layers](#).

Goal: decouple the services or shards, remove the performance bottleneck (*Shared Database*).

Prerequisite: the domain data is weakly coupled.

If the data clearly follows subdomains, it may be possible to subdivide it accordingly. The services will become [choreographed](#) (or [orchestrated](#) if they have an [integration layer](#)) instead of communicating through the [shared data](#).

Pros:

- The services become independent in their persistence and data processing technologies.
- Performance of the *data layer*, which tends to limit the scalability of the system, will likely improve thanks to the use of smaller specialized databases.

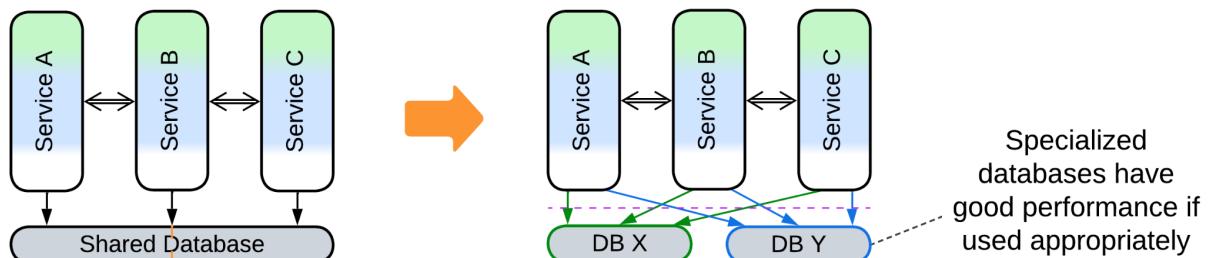
Cons:

- The communication between the services and the synchronization of their data becomes a major issue.
- Joins of the data from different subdomains will not be available.
- Costs are likely to increase because of data transfer and duplication between the services.
- You will have to administrate multiple databases.

Further steps:

- [CQRS Views \[MP\]](#) or a [Query Service \[MP\]](#) help a service access and join data that belongs to other services.

Deploy specialized databases



Patterns: [Polyglot Persistence](#).

Goal: improve performance and maybe fault tolerance of the data layer.

Prerequisite: there are diverse data types or patterns of data access.

It is very likely that you can either use [specialized databases](#) for various data types or deploy [read-only replicas](#) of your data for analytics.

Pros:

- You can choose one of the many specialized databases available on the market.
- There is a good chance to significantly improve performance.
- Replication improves fault tolerance of your data layer.

Cons:

- It may take effort to learn the new technologies and use them efficiently.
- Someone needs to see to the new database(s).
- You'll likely need to work around *replication lag* [[MP](#)].

Proxy:

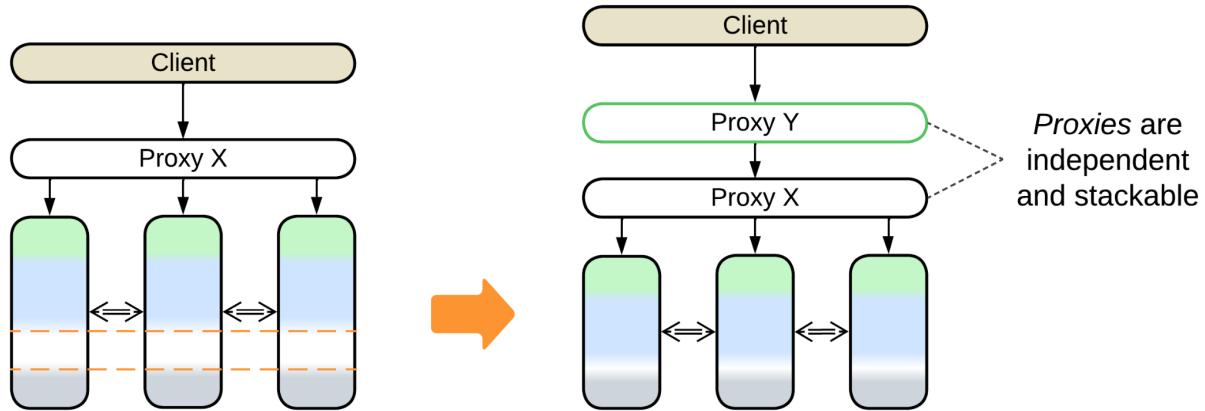
It usually makes little sense to get rid of a [Proxy](#) once it is integrated into a system. Its only real drawback is a slight increase in latency for user requests which may be helped through creation of [bypass channels](#) between the clients and a service that needs low

latency. The other drawback of the pattern, the *Proxy's* being a single point of failure, is countered by deploying multiple instances of the *Proxy*.

As *Proxies* are usually third-party products, there is very little we can change about them:

- We can add another kind of a *Proxy* on top of the existing one.
- We can use a stack of *Proxies* per client, making [Backends for Frontends](#).

Add another Proxy



Patterns: [Proxy](#), [Layers](#).

Goal: avoid implementing generic functionality.

Prerequisite: you don't have this kind of *Proxy* yet.

A system is not limited to a single kind of *Proxies*. As a *Proxy* represents your system without changing its function, *Proxies* are transparent, thus they are stackable.

It often makes sense to colocate software *Proxies* or use a multifunctional *Proxy* to reduce the number of network hops between the clients and the system. However, in a highly loaded system *Proxies* may be resource-hungry, thus in some cases colocation strikes back.

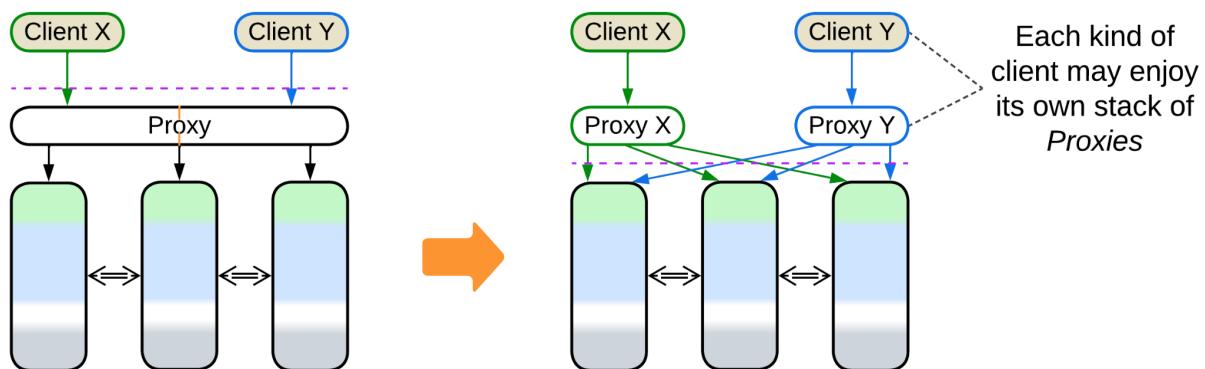
Pros:

- You get another aspect of your system implemented for you.

Cons:

- Latency degrades.
- More work for admins.
- Another point of possible failure.

Deploy a Proxy per client type



Patterns: [Proxy](#), [Backends for Frontends](#).

Goal: let the aspects of communication vary among kinds of clients.

Prerequisite: your system serves several kinds of clients.

If you have internal and external clients, or admins and users, you may want to vary the setup of *Proxies* for each kind of client, sometimes to the extent of physically separating network communication paths, so that each kind of client is treated according to its bandwidth, priority and permissions.

Pros:

- It is easy to set up various aspects of communication for a group of clients.

Cons:

- More work for admins as the *Proxies* are duplicated.

Orchestrator:

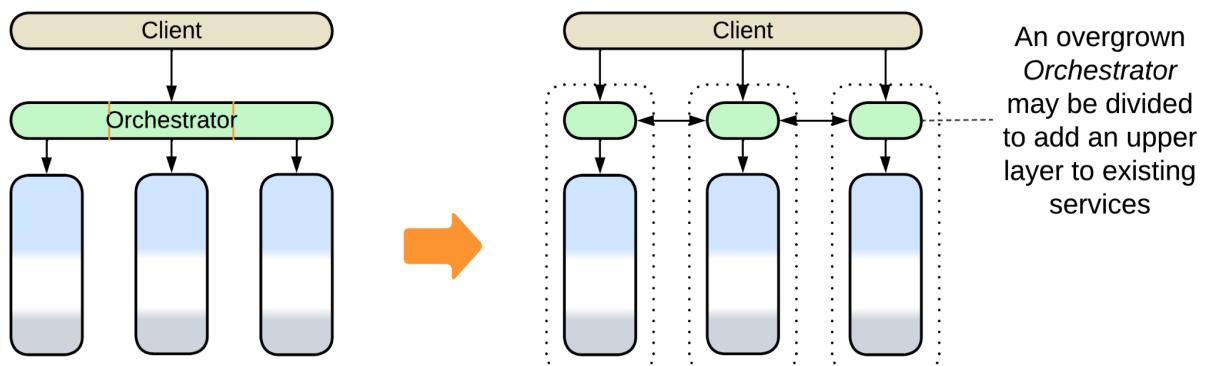
Employing an *Orchestrator* has two pitfalls:

- The system becomes slower because too much communication is involved.
- A single *Orchestrator* may be found to be too large and rigid.

There is one way to counter the first point and more ways to solve the second one:

- Subdivide the *Orchestrator* by the system's subdomains, forming [Layered Services](#) and minimizing network communication.
- Subdivide the *Orchestrator* by the type of client, forming [Backends for Frontends](#).
- Add another [layer](#) of orchestration.
- Build a [Top-Down Hierarchy](#).

Subdivide to form Layered Services



Patterns: [Orchestrated Three-Layered Services \(Layered Services \(Services, Layers\)\)](#).

Goal: simplify the *Orchestrator*, let the service teams own orchestration, decouple forces for the services, improve performance.

Prerequisite: the high-level ([orchestration](#)) logic is weakly coupled between the subdomains.

If the *orchestration* logic mostly follows the subdomains, it may be possible to subdivide it accordingly. Each service gets a part of the *Orchestrator* that mostly deals with its subdomain but may call other services when needed. As a result, [each service orchestrates every other service](#). Still, a large part of orchestration becomes internal to the service, meaning that fewer calls over the network are involved.

Pros:

- You subdivide the large *Orchestrator* codebase.

- Performance is improved.
- The services become more independent in their quality attributes.

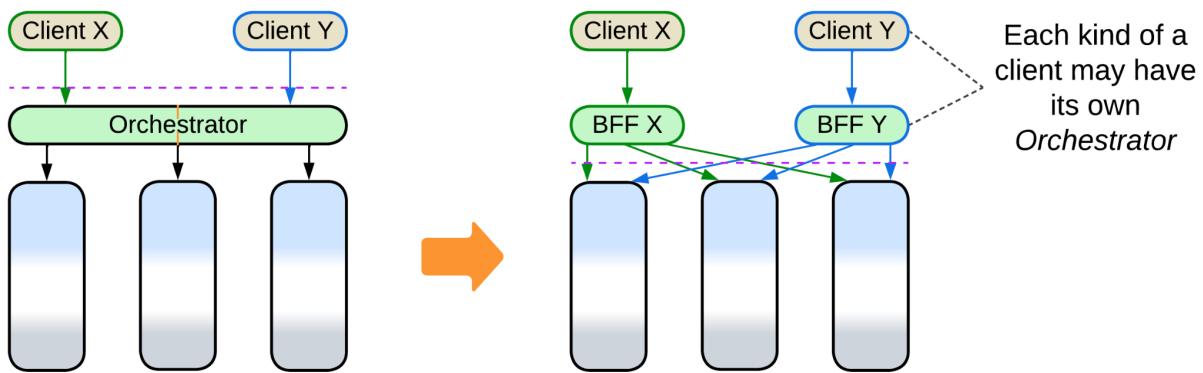
Cons:

- You lose the client-facing *orchestration team* – now each service's team will need to face its clients.
- Service teams become interdependent (while having equal rights), which may result in slow development and suboptimal decisions.
- There is no way to share code between different use cases or even take a look at all of the scenarios at once.

Further steps:

- [CQRS Views \[MP\]](#) or a [Query Service \[MP\]](#) help a service access and join data that belongs to other services, further reducing the need for interservice communication.

Subdivide to form Backends for Frontends



Patterns: [Backends for Frontends](#), [Orchestrator](#).

Goal: simplify the *Orchestrator*, employ a team per client type, decouple qualities for clients.

Prerequisite: clients vary in workflows and forces.

When use cases for clients vary, it makes sense for each kind of client to have a dedicated *Orchestrator*.

Pros:

- The smaller *Orchestrators* are independent in qualities, technologies and teams.
- The smaller *Orchestrators* are ... well, smaller.

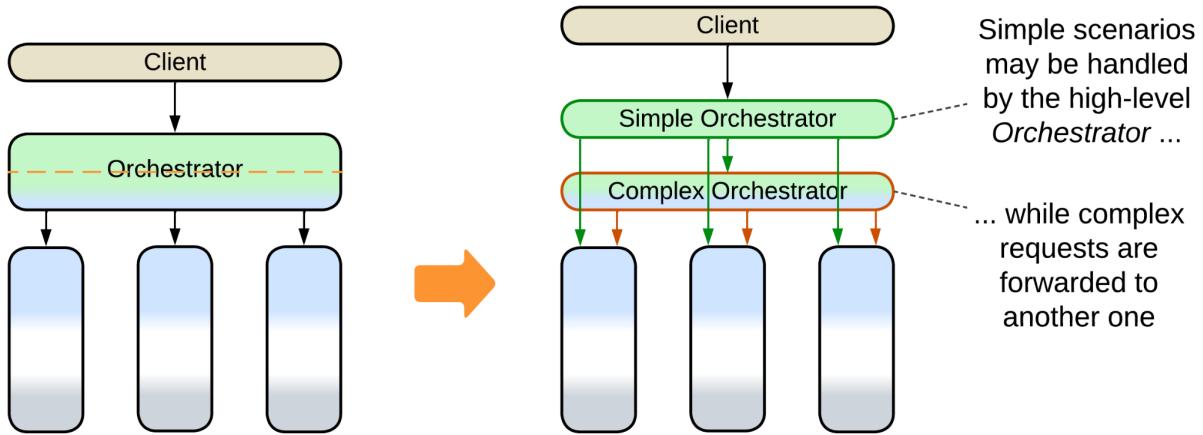
Cons:

- There is no good way to [share code](#) between the *Orchestrators*.

Further steps:

- You may want to add client-specific [Proxies](#) and, maybe, co-locate them with the *Orchestrators* to avoid the extra network hop.
- Adding another shared [Orchestrator](#) below the ones dedicated to clients creates a place for sharing functionality among the *Orchestrators*.
- If you are running [Microservices](#) over a [Service Mesh](#), [Sidecars](#) [DDS] may help to share generic code.

Add a layer of orchestration



Patterns: [Orchestrator](#), [Layers](#).

Goal: implement simple use cases quickly, while still supporting complex ones.

Prerequisite: use cases vary in complexity.

You may use two or three *orchestration frameworks* (engines) which differ in complexity. A simple declarative tool may be enough for the majority of user requests, reverting to custom-tailored code for rare complex cases.

Pros:

- Simple scenarios are easy to write.
- You retain good flexibility with hand-written code when it is needed.

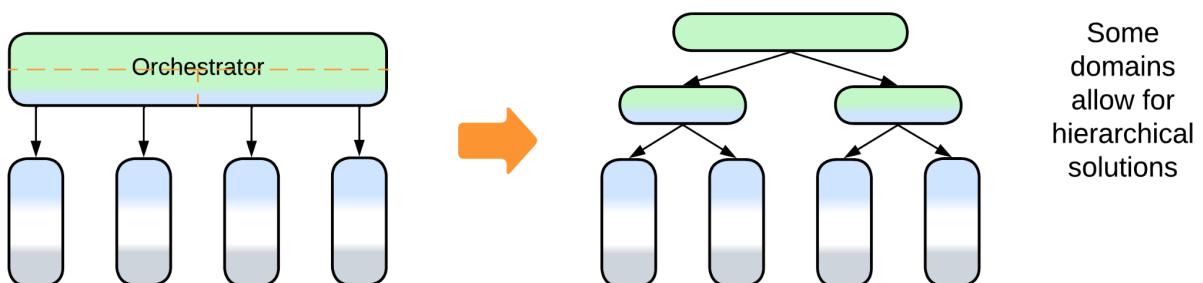
Cons:

- Requires learning multiple technologies.
- More components mean more failures and more administration.
- Performance of complex requests may suffer from more indirection.

Further steps:

- Divide one or more of the resulting *orchestration layers* to form [Layered Services](#), [Backends for Frontends](#), [Hierarchy](#), or [Cell-Based Architecture](#).

Form a hierarchy



Patterns: [Top-Down Hierarchy \(Hierarchy\)](#).

Goal: simplify the *Orchestrator* and, if possible, the services.

Prerequisite: the domain is hierarchical.

If an *Orchestrator* becomes too complex, some domains (e.g. IIoT or telecom) encourage using a tree of *Orchestrators*, with each layer taking care of one aspect of the domain, serving the most generic functionality at the root.

Pros:

- Multiple specialized teams and technologies.
- Small codebase per team.

- Reasonable testability.
- Some decoupling of quality attributes.

Cons:

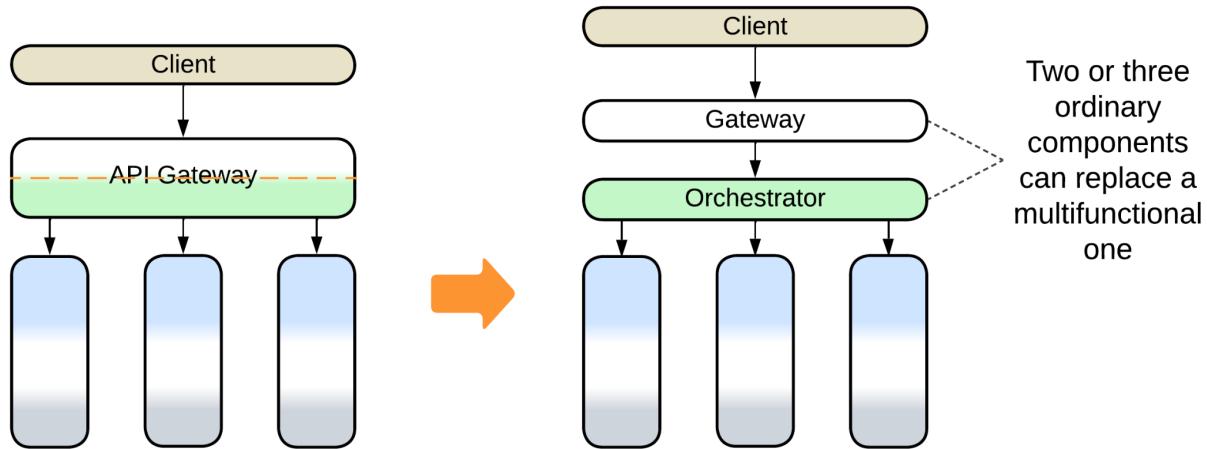
- Hard to debug.
- Poor latency in global scenarios unless several layers of the *hierarchy* are colocated.

Combined Component:

The patterns that involve *orchestration* ([API Gateway](#), [Event Mediator](#), [Enterprise Service Bus](#)) may allow for [most of the evolutions](#) of the [Orchestrator](#) metapattern by deploying multiple versions of the *Combined Component* that differ in their orchestration logic. There is also a special evolution:

- Replace the *Combined Component* with several specialized ones

Divide into specialized layers



Patterns: [Layers](#).

Goal: break out of *vendor lock-in* [[DDD](#)], gain flexibility.

Prerequisite: you have lots of free time.

If you feel that the *Combined Component* which your system relies on does not cover all your needs, or is too expensive, or unstable, then you may want to get rid of it by replacing it with generic single-purpose tools or a homebrewed implementation that will always adapt to your circumstances.

Pros:

- It's free.
- You'll own your code.
- Anything you write will fit your needs for as long as you spend time supporting it.

Cons:

- Takes lots of work.
- Performance may become worse because there will be more components on the requests' path and also because the industry-grade framework that you used could have been highly optimized.

Appendix F. Format of a metapattern.

The descriptions of most metapatterns follow the same format:

Diagram

The structural diagram (in *abstractness-subdomain-sharding coordinates*) of a typical application of the metapattern. Please note that in practice the number and types of components and their interactions may vary:

- Even though most diagrams show 3 *layers* or services, there are many 2-layered or 4-layered systems, while the number of services may often be greater than 10.
- *Extension metapatterns* add a *layer* (or a *layer of services*) to an existing system, which is shown as *Services*, but may instead comprise *Shards*, *Layers* or even a *Monolith*.
- Subtypes of *Hierarchy* or *Mesh* differ in their topologies. Only one is shown.
- Components of metapatterns may communicate in various ways that include in-process calls, RPC, asynchronous messaging or streams. Only one of them is shown. Optional communication pathways may appear as dashed arrows.

Most diagrams feature the following colors:

- *Use cases* (aka integration, orchestration, workflow or application logic) are shown in green. Those are high-level scenarios executed by user actions or signals from hardware which keep the system acting as a whole. Use cases are *what* your software does.
- *Domain logic* (business rules), shown in blue, is the set of algorithms that models the real-world system your software describes. It is *how* your system solves its tasks.
- *Generic code* is white. It stands for tools and libraries unrelated to your business. Examples include communication protocols, data compression and common maths.
- *Data* is gray. It includes business-critical in-memory state (e.g. user's session) and persistent storage (in a database or files).

Use cases and *domain logic* comprise *business logic* [[PEAA](#)] – the code that makes your software different from whatever else is on the market. It is this part of the system which your customers pay for, and it usually is much larger than the other parts, which makes business logic the primary focus of development.

In *choreographed* systems use cases are defined by the web of communication channels instead of code inside the system's components. That is represented by green arrows and overall lack of green areas on corresponding diagrams.

Abstract

Motto and the design goal.

Known as: the list of aliases for the general metapattern.

Aspects: an optional list of roles the subject component may have.

Variants or *examples*: one or more lists of notable variations in, patterns or common architectures that derive from or implement the subject metapattern.

Structure: a short description of the structure of the metapattern.

Type: Root, main, extension or implementation.

- The *root* of all the metapatterns is [*Monolith*](#), as any system both looks monolithic to its clients and comes about through division of the continuous (monolithic) design space.
- Main metapatterns ([*Layers*](#), [*Services*](#) and few others derived from them) stay at the core of any architecture.
- An *extension* adds components to an architecture, built around a main metapattern, to modify its properties.
- An *implementation* metapattern shows the internal structure of a component which is usually treated as monolithic.

A short *table of benefits and drawbacks*.

References: select articles and books that describe the topic.

After that, follow two or three paragraphs of facts and ideas about the metapattern .

Performance

This section discusses the performance of the subject metapattern in scenarios which vary in their extent: simple requests or events that relate to a single subsystem are usually processed much faster than those that touch multiple components.

There are two kinds of performance: latency and throughput. Low latency is possible only if few components are involved because inter-component communication, especially in distributed systems, increases latency. Contrariwise, throughput depends on the number of components that work in parallel, thus it scales together with the system.

This section may also discuss optimization techniques that apply to the metapattern.

Dependencies

Some components of the metapattern depend on other components. If a component changes, everything that depends on it may need to be re-tested with the updated version. If a component's interface changes, all the components that depend on it must be updated. Therefore, components that evolve quickly should depend on others, not the other way around.

Some patterns, like [*Hexagonal Architecture*](#), use [*Adapters*](#) to break dependencies. An *Adapter* depends on components on both sides of itself, making those components independent of each other. The *Adapters* are small enough to update quickly and may easily be replaced with stubs for testing or running a component in isolation.

Applicability

Here follows a list of types of projects which may benefit from applying the architecture under review, and another list of those which it is more likely to hurt.

Relations

These are shown through an optional sequence of diagrams, showing the (*extension* or *implementation*) metapattern applied to various kinds of architectures, followed by a list of relations between the current and other metapatterns.

Variants and examples

A metapattern usually unites many variations of several patterns. Here we may have a section per dimension of variability and a section for well-known examples of the pattern.

On some occasions I had to include several variants that do not properly belong to the metapattern under review, just to avoid confusion with terminology and point the reader to a right chapter. For example, [*Modular Monolith*](#) has a module per subdomain, thus it belongs to [*Services*](#) rather than [*Monolith*](#). Still, when the chapter on *Monolith* was not mentioning it, I was blamed for misunderstanding the monolithic architecture. Such patterns are marked as (misapplied) or (inexact).

I tried to show the difference between synonymous names for every variant or example whenever I could identify one.

Evolutions

This covers a brief summary of possible changes to the architecture under review. Each change leads to a new architecture which usually matches another metapattern.

[Appendix E](#) discusses many evolutions in greater detail:

- A diagram that shows the original and resulting structure.
- The list of patterns, present in the resulting architecture. More general forms of each pattern are given in parentheses, i.e. Pattern (Metapattern (Parent Metapattern)).
- The goal(s) of the transition.
- The prerequisites that enable the change.
- A short description of the change and the resulting system.
- Lists of pros and cons of the evolution.
- An optional list of metapatterns that the resulting system may evolve into and their benefits in the context of the current evolution.

Appendix G. Glossary.

Abstractness – the scope of information that a *concept* operates. Highly abstract concepts describe the system's behavior in less words.

Action – an act of a system that changes its environment.

API (application programming interface) – a set of *methods* or *messages* that a component exposes to its *clients*.

Application – the most abstract layer that usually *integrates components* of a less abstract layer.

Architectural pattern – a way to structure a system or a part of a system to achieve desirable properties (address a set of *forces*).

Architectural style – see *architecture*.

Architecture – the structure of a system. It comprises *components* and their *interactions*.

ASS diagram – a *structural diagram* with *abstraction*, *subdomain* and *sharding* for coordinates.

Asynchronous communication – the mode of *communication* when the sender of the *request message* does not stop the execution of its *scenario* to wait for the confirmation message.

Attack surface – the amount of *components* and functionality that faces an external network (potentially exposed to hackers).

Availability – the percentage of time that the system is operational (satisfies its *users*).

Bounded context – a subset of *requirements* and code that shares a set of *concepts*. Usually consists of internals of a *component* and *APIs* of all the *components* it uses.

Business logic – the thing that *users* pay for. It is the heart of the business and is usually the largest part of the *project*. You cannot buy *business logic*, only *implement* it. *Business logic* comprises *use cases* and *business rules*.

Business rules – domain concepts and their relations. They make the low-level half of *business logic*.

Choreography – a kind of *workflow* in which *components* that belong to the same level of *abstractness* cooperate to implement a *use case*.

Client – an external *component* or *system* that makes use of a *component* or *system* in question.

Cohesion – the density of logical connections between entities inside a *component*.

Colocated – running in the same address space (process) on the same hardware.

Communication – transfer of data or signals in a *system*.

Complexity – the cognitive load caused by the quantity of entities (*concepts* or *modules*) and their relations that a programmer needs to operate.

Component – an encapsulated part of a *system*. It exposes an *API* to the system's *clients* and/or other *components* of the *system*.

Concept – a notion of an element of a *system's* behavior, usually present in *requirements*.

Contract – the informal rules for the behavior of a *component* expected by its *clients*.

Control – a kind of system that supervises physical entities or external programs.

Coupling – the density of logical connections between *components*.

Cross-cutting concern – a functionality that should be present in multiple *components*.

Debugging – trying to force the *system* to behave correctly from the user's point of view.

Deployment – uploading a *component* to the hardware that will execute it.

Design – planning for the best way to write code.

Design – see *architecture*.

Design space – the multitude of possible ways to *design* a given project.

Development – building a *project* for its *users*. Usually involves intermixed *design*, *implementation*, *debugging* and *testing* phases.

Distributed – spread over multiple computers that *communicate* via a network.

Domain – the whole of knowledge (including *requirements*) that is needed to build a *system*.

Domain – the middle layer of a *system* that contains its *business rules*.

Event – a signal that carries some meaning for a *system* or *component*. *Events* may carry data.

Fault tolerance – the ability of a *system* to remain (at least partially) operational if one or more of its *components* fail (become inaccessible due to a hang, crash or a hardware failure).

Forces – expected properties of a *system* (such as its stability or response time) which are crucial for the *system* to be built, *deployed* and used successfully.

Functional requirements – the *requirements* that describe *inputs* and *outputs* of a *system*, but not its *performance* or stability.

Global use case – a *use case* that involves most of the *components* of a *system*. Such scenarios are strongly affected by the *system*'s structure.

Implementation – the process of writing code.

Implementation – internals of a *component*.

Infrastructure – the lowest layer of a *system* that provides general-purpose functionality (tools) to its upper layers.

Input – events or data that a *system* reacts to.

Integration – see *orchestration*.

Integration complexity – the *complexity* of understanding how individual *components* interact to make a *system*.

Interactions – the kinds and routes of *communication* between *components* of a *system*.

Interface – see *API*.

Latency – the delay between a *system*'s receiving *input* and producing a corresponding *output*.

Layers – *components* of a *system* partitioned by the level of *abstraction*.

Messaging – communication by sending short pieces of data.

Method call – invocation of an *interface* method (or procedure) of a *component* by another *component*.

Metapattern – a cluster of *patterns* that have similar *structural diagrams* and address related issues.

Module – a colocated (in-process) *component*.

Non-functional requirements (NFRs) – see *forces*.

Notification – an event that one *component* sends to another *component*(s) to inform them of some change.

Operational complexity – see *integration complexity*.

Orchestration – a kind of *workflow* where a single dedicated *component* (*Orchestrator*) makes use of (usually multiple) less abstract *components*. *Facade* [[GoF](#)] is a good example.

Output – actions or data that a *system* produces.

Pattern – a documented approach (blueprint) for solving a recurrent programming issue.

Pattern Language – a set of interrelated *patterns* intended to cover most aspects of designing systems in a target *domain*.

Performance – a measure of a system's *throughput*, *latency* and *resource consumption*.

Persistent data – data which survives rebooting the software.

Pipeline – a set of *components* for stepwise processing of data.

Processing – transformation of *input* data into *output* data.

Project – the process of making a system.

Pub/sub (publish/subscribe) – a mode of communication when one *component (subscriber)* receives a subset of *notifications* from another *component (publisher)*. It is the *subscriber* that chooses which *notifications* it is interested in.

Qualities – the properties which a *component* or (sub)system manifests to satisfy *forces*.

Real-time – a force that requires the system to respond to incoming events immediately.

Request/confirmation – a pair of messages between two components (Requestor and Executor). The *request* describes an *action* that the requestor wants the executor to run ($R \Rightarrow E$). The *confirmation* describes the results of the execution ($R \leq E$).

Requirements – a set of rules that describe the correct (expected) behavior of the system.

Resources – CPU, memory, network bandwidth and other stuff that costs money.

Scaling – ability to increase *throughput* of a system by providing it with more *resources*.

Scenario – see *use case*.

Service – a distributed component.

Services – components of a system partitioned by *subdomain*.

Sharding – deploying multiple instances of a *component*.

Single point of failure – a software or hardware *component* which if fails makes the entire system non-operational. High-availability systems should avoid *single points of failure*.

SPI (service provider interface) – a set of methods or messages that a *component* expects to be supported by the *components* it uses.

State – data that a *component* keeps between processing its *inputs*.

Structural diagram – a graphical representation of the structure of a (sub-)system that shows *components* and their *interactions*.

Stub – a very simple implementation of a *module* that allows other *components* that use it to run without starting the original *module*. *Stubs* are used to implement modules concurrently or test them in isolation.

Subdomain – a distinct cohesive part of *domain knowledge*.

Synchronous communication – the mode of communication when the requesting component waits for results of its *request* to another *component* before continuing to run its task.

System – a self-sufficient set of communicating components that were brought together or implemented to satisfy its users (by running *use cases*).

Task – a high-level sequence of execution steps. Similar to *use case* or *scenario*.

Team – a few programmers and testers that work on a *component*. Teams of more than 5 members lose productivity to communication overhead.

Testing – checking how satisfactorily the system behaves.

Throughput – the amount of data a system can process per unit of time.

Use case – a behavior expected by system's users. A system is implemented to run *use cases*. *Use cases* are the high-level half of *business logic*.

User – a human that uses a system and usually pays well if satisfied with its behavior.

Vendor lock-in – a pitfall when a system relies on an external provider so much that it is impossible to change the provider. It is similar to falling prey to a monopoly.

Workflow – a sequence of actions (*messages or method calls*) required to *implement a use case*.

Appendix H. History of changes.

0.1 (2020) – Description of my semisynchronous *Proactor* architecture for a VoIP gateway, published by dou.ua. It received very positive feedback and lots of comments from the community.

0.2 (2020) – [The same in a more official style](#) for the (Corona-)PLoP'20 conference.

0.3 (2021) – Comparison of choreography and orchestration for dou.ua. No impact.

0.4 (2022) – A series of 5 articles that looked into local and distributed architectures by applying the actor model. Positive feedback from dou.ua, but the series was interrupted by the war.

0.5 (2023) – [The same series in English](#), published by ITNEXT and upvoted by r/softwarearchitecture.

0.6 (2023) – I attempted to rebuild the series for InfoQ but the first article was rejected as impractical (technology-agnostic).

0.7 (09-2024) – [Chapters from this book](#), published by ITNEXT. Some of them were boosted by Medium.

0.8 (11-2024) – The complete book as a pdf. Clients were changed to mid-brown. Detailed evolutions were moved to the appendix. Rejected by Manning (the free license and color diagrams make the book unprofitable) and O'Reilly (it would get in the way of their bestsellers). Ignored by Addison-Wesley.

0.9 (12-2024) – Integrated patterns from [[DDS](#), [LDDD](#), [SAHP](#)] and Internet sources, mostly affecting [Shards](#), [Pipeline](#), [Proxy](#), [Orchestrator](#) and [Hexagonal Architecture](#). Added diagrams for [Polyglot Persistence with derived storage](#) and detailed evolutions for [Pipeline](#). Downgraded [analytical chapters](#) to sections and added a couple of new ones. Extended the [ambiguous patterns chapter](#). Improved the structure of the variants sections of metapatterns: now each synonym has a short description. Fixed alignment of text and figures. Liked by [r/softwarearchitecture](#). Rejected by The Pragmatic Programmer (they want “hands-on, actionable content”). Ignored by No Starch Press and Packt.

1.0 (04-2025) – Integrated [[DEDS](#)]. Integration logic (use cases) is now in green. Added [MVC-related patterns](#) and a section on [Programming and architectural paradigms](#). Replaced the chapter on [control and processing](#) with a new one about [Four kinds of software](#) and added another one called [The heart of software architecture](#). Made minor changes all over the book. Now I know how to generate a table of contents for both EPUB and PDF versions. Ignored by Wikibooks.

1.1 (07-2025) – Lars Noodén edited the book, fixing my poor English. Patterns are now in *Title Case Italics*. [Domain-Oriented Microservice Architecture](#) was added. There are now short explanation sections (in gray) throughout the book.

Appendix I. Index of patterns.

[Action-Domain-Responder](#) (ADR)
[Actors](#) (architecture)
[Actors](#) (as Mesh)
[Actors](#) (backend)
[Actors](#) (embedded systems)
[Actors](#) (scope)
[Adapter](#)
[Addons](#)
[Aggregate Data Product Quantum](#) (Data Mesh)
[Ambassador](#)
[Anticorruption Layer](#)
[API Composer](#)
[API Gateway](#)
[API Gateway](#) (as Orchestrator)
[API Gateway](#) (as Proxy)
[API Rate Limiter](#)
[API Service](#) (adapter)
[API Throttling](#)
[Application Layer](#) (Orchestrator)
[Application Service](#)
[Aspects](#) (Plugins)
[Atomically Consistent Saga](#)
[Automotive SOA](#) (as Service-Oriented Architecture)
[AUTOSAR Classic Platform](#) (as Microkernel)
[Backend for Frontend](#) (adapter)
[Backends for Frontends](#) (BFF)
[Batch Processing](#)
[Big Ball of Mud](#)
[Blackboard](#)
[Bottom-Up Hierarchy](#)
[Broker](#) (Middleware)
[Broker Topology Event-Driven Architecture](#)
[Bus of Buses](#)
[Cache](#) (read-through)
[Cache-Aside](#)
[Caching Layer](#)
[Cell](#) (WSO2 definition)
[Cell Gateway](#) (WSO2 Cell-Based Architecture)
[Cell Router](#) (Amazon Cell-Based Architecture)
[Cell-Based Architecture](#) (WSO2 version)
[Cell-Based Microservice Architecture](#) (WSO2 version)
[Cells](#) (Amazon definition)
[Choreographed Event-Driven Architecture](#)
[Choreographed Two-Layered Services](#)

[Clean Architecture](#)
[Cluster](#) (group of services)
[Combined Component](#)
[Command Query Responsibility Segregation](#) (CQRS)
[Composed Message Processor](#)
[Configuration File](#)
[Configurator](#)
[Container Orchestrator](#)
[Content Delivery Network](#) (CDN)
[Control](#) (Orchestrator)
[Controller](#) (Orchestrator)
[Coordinator](#) (Saga)
[CQRS View Database](#)
[Create on Demand](#) (temporary instances)
[Data Archiving](#)
[Data Domain](#)
[Data File](#)
[Data Grid](#) (Space-Based Architecture)
[Data Lake](#)
[Data Mesh](#)
[Data Product Quantum](#) (DPQ)
[Data Warehouse](#)
[Database Cache](#)
[Database Abstraction Layer](#) (DBAL or DAL)
[Dependency Inversion](#)
[Deployment Manager](#)
[Device Drivers](#)
[Direct Server Return](#)
[Dispatcher](#) (Proxy)
[Distributed Cache](#)
[Distributed Middleware](#)
[Distributed Monolith](#)
[Distributed Runtime](#) (client point of view)
[Distributed Runtime](#) (internals)
[Document-View](#)
[Domain](#) (Uber definition for WSO2-style Cell)
[Domain-Driven Design](#) (layers)
[Domain-Oriented Microservice Architecture](#) (DOMA)
[Domain Services](#) (scope)
[Domain-Specific Language](#) (DSL)
[Edge Service](#)
[Embedded systems](#) (layers)
[Enterprise Service Bus](#) (ESB)
[Enterprise Service Bus](#) (as Middleware)
[Enterprise Service Bus](#) (as Orchestrator)
[Enterprise Service-Oriented Architecture](#)
[Enterprise SOA](#)
[Event Collaboration](#)

[Event-Driven Architecture](#) (EDA)
[Event Mediator](#)
[Event Mediator](#) (as Middleware)
[Event Mediator](#) (as Orchestrator)
[Event-Sourced View](#)
[Eventually Consistent Saga](#)
[External Search Index](#)
[FaaS](#)
[FaaS](#) (pipelined)
[Facade](#)
[Firewall](#)
[Flavors](#) (Plugins)
[Front Controller](#) (query service of a pipeline)
[Full Proxy](#)
[Function as a Service](#)
[Game Development Engine](#)
[Gateway](#) (adapter)
[Gateway Aggregation](#)
[Grid](#)
[Half-Proxy](#)
[Half-Sync/Half-Async](#)
[Hardware Abstraction Layer](#) (HAL)
[Hexagonal Architecture](#)
[Hexagonal Service](#)
[Hierarchical Model-View-Controller](#) (HMVC)
[Hierarchy](#)
[Historical Data](#)
[Hooks](#) (Plugins)
[Hypervisor](#)
[In-Depth Hierarchy](#)
[Ingress Controller](#)
[Instances](#)
[Integration Database](#)
[Integration Service](#)
[Integration Microservice](#)
[Interpreter](#)
[Layered Architecture](#)
[Layered Microservice Architecture](#) (Backends for Frontends)
[Layered Monolith](#)
[Layered Service](#)
[Layered Services](#) (architecture)
[Layers](#)
[Leaf-Spine Architecture](#)
[Load Balancer](#)
[MapReduce](#)
[Materialized View](#)
[Mediator](#)
[Memory Image](#)

[Mesh](#)
[Message Broker](#)
[Message Bus](#)
[Message Bus \(as Middleware\)](#)
[Message Translator \(adapter\)](#)
[Messaging Grid \(Space-Based Architecture\)](#)
[Microgateway](#)
[Microkernel](#)
[Microkernel \(Plugins\)](#)
[Microkernel Architecture \(Plugins\)](#)
[Microservices \(architecture\)](#)
[Microservices \(scope\)](#)
[Middleware](#)
[Model 1 \(MVC1\)](#)
[Model 2 \(MVC2\)](#)
[Model-View-Adapter \(MVA\)](#)
[Model-View-Controller \(MVC\)](#)
[Model-View-Presenter \(MVP\)](#)
[Model-View-ViewModel \(MVVM\)](#)
[Modular Monolith](#)
[Modulith](#)
[Monolith](#)
[Monolithic Service](#)
[Multitier Architecture](#)
[Multi-Worker](#)
[Nanoservices \(API layer\)](#)
[Nanoservices \(as runtime\)](#)
[Nanoservices \(pipelined\)](#)
[Nanoservices \(scope\)](#)
[Nanoservices \(SOA\)](#)
[Native Data Product Quantum \(sDPQ\)](#)
[Nearline System](#)
[Network of Networks](#)
[N-Tier Architecture](#)
[Offline System](#)
[Onion Architecture](#)
[Open Host Service](#)
[Operating System](#)
[Operating System Abstraction Layer \(OSAL or OAL\)](#)
[Orchestrated Saga](#)
[Orchestrated Services](#)
[Orchestrated Three-Layered Services](#)
[Orchestrator](#)
[Orchestrator of Orchestrators](#)
[Partition](#)
[Peer-to-Peer Networks](#)
[Persistent Event Log](#)
[Pipeline](#)

[Pipes and Filters](#)
[Platform Abstraction Layer](#) (PAL)
[Plug-In Architecture](#)
[Plugins](#)
[Polyglot Persistence](#)
[Ports and Adapters](#)
[Pool](#) (stateless instances)
[Presentation-Abstraction-Control](#) (PAC)
[Proactor](#)
[Process Manager](#)
[Processing Grid](#) (Space-Based Architecture)
[Proxy](#)
[Published Language](#)
[Query Service](#)
[Rate Limiter](#)
[Reactor](#) (multi-threaded)
[Reactor](#) (single-threaded)
[\(Re\)Actor-with-Extractors](#)
[Read-Only Replica](#)
[Read-Through Cache](#)
[Reflection](#) (Plugins)
[Remote Facade](#)
[Replica](#)
[Replicated Cache](#)
[Replicated Stateless Services](#) (instances)
[Reporting Database](#)
[Repository](#)
[Request Hedging](#)
[Response Cache](#)
[Resource-Method-Representation](#) (RMR)
[Reverse Proxy](#)
[Saga Engine](#) (Microkernel)
[Saga Execution Component](#)
[Saga Orchestrator](#)
[Scaled Service](#)
[Scatter-Gather](#)
[Scheduler](#)
[Script](#)
[Segmented Microservice Architecture](#)
[Separated Presentation](#)
[Service-Based Architecture](#) (architecture)
[Service-Based Architecture](#) (shared database)
[Service Layer](#) (Orchestrator)
[Service Mesh](#)
[Service Mesh](#) (as Mesh)
[Service Mesh](#) (as Middleware)
[Service of Services](#)
[Service-Oriented Architecture](#) (SOA)

[Services](#)

[Services of Services](#)

[Sharding](#) (persistent slices of data)

[Sharding Proxy](#)

[Shards](#)

[Shared Database](#)

[Shared Databases](#) (Polyglot Persistence)

[Shared Event Store](#)

[Shared File System](#)

[Shared Memory](#)

[Shared Repository](#)

[Sidecar](#)

[Software Framework](#) (Microkernel)

[Source-Aligned Data Product Quantum](#) (Data Mesh)

[Space-Based Architecture](#) (as Mesh)

[Space-Based Architecture](#) (as Middleware)

[Specialized Databases](#)

[Spine-Leaf Architecture](#)

[Stamp Coupling](#)

[Strategy](#) (Plugins)

[Stream Processing](#)

[Three-Tier Architecture](#)

[Tiers](#)

[Top-Down Hierarchy](#)

[Transaction Script](#)

[Virtualizer](#)

[Work Queue](#)

[Workflow System](#)

[Workflow Owner](#) (Orchestrator)

[Wrapper Facade](#) (Orchestrator)

[Write-Behind Cache](#)

[Write-Through Cache](#)