# Architecture Recommendations

# Business drivers for architectural decisions - “Why”

Statistical organisations are constantly trying to evolve their IT-systems to better fulfill their business needs or to achieve new business goals. In some cases the drive for change is triggered by external phenomenons such as Big Data/new data sources or the need for higher business agility in order to align more quickly to new user demands. In other cases, it is the technological advancements such as SOA or cloud computing that enables or requires a change in how IT-systems are built. Few business processes, and their supporting IT-systems, live in isolation and therefore the way that these IT-systems are connected is an important consideration in order to keep development and governance costs low and to enable organisations to more quickly change to meet new requirements.

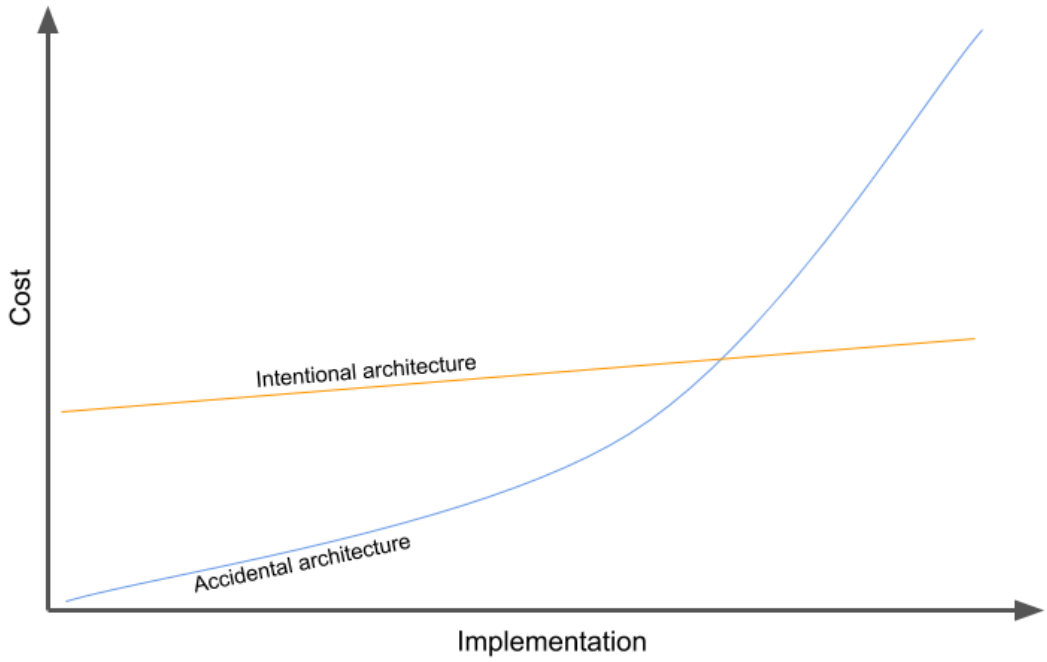
Some of the commonly occurring business drivers within statistical organisations that requires architectural decisions to be made are:

- *Transitioning from subject matter oriented software to shared process oriented software*
- *Establishing better data management on enterprise level*
- *Reusing services created by other statistical organisations*

Not all statistical organisations focus on all of these business drivers and there could be multiple other important business drivers that should also influence architectural decisions so this document should be seen as a baseline of guidance and not as a complete guidance for architectural decision making.

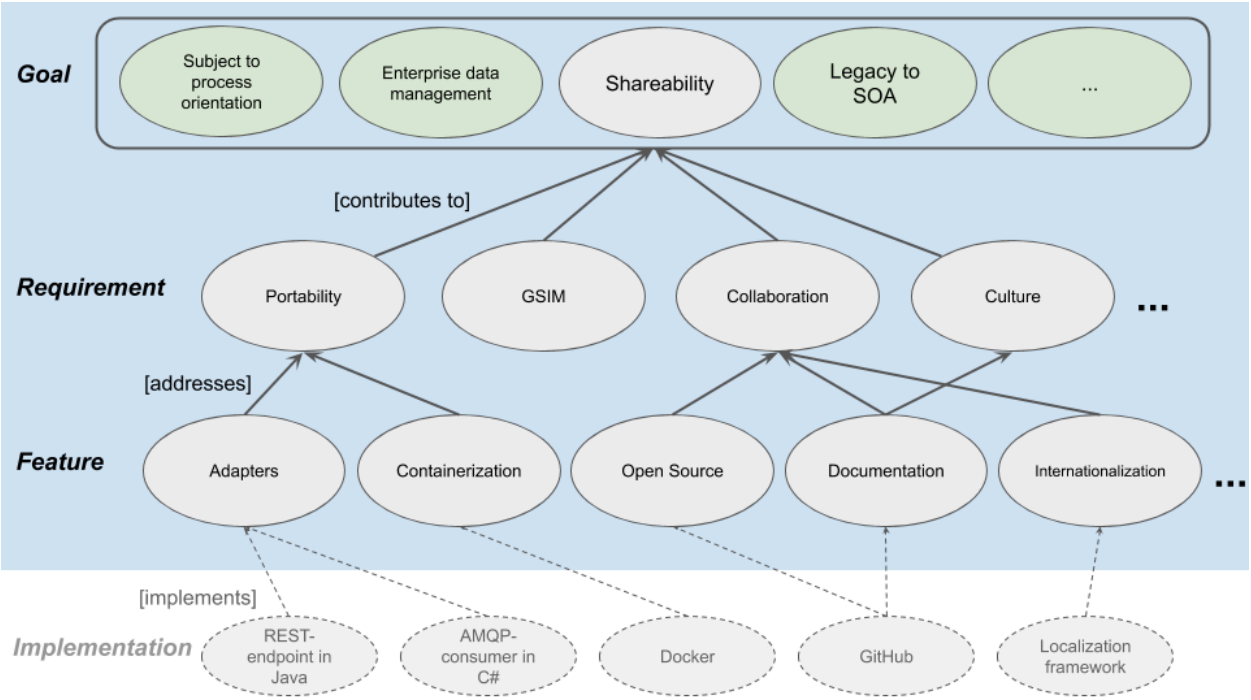
The concept of architecture in this document involves principles and patterns that allows for a more coherent, consistent approach when designing both IT-systems and the environment that these exist in, with the purpose of not only meeting the requirements from the end users but also to enable the organisation to reach its long term goals. Therefore, this guidance is more focused on what consequences different patterns can have on the sum of all IT-systems in the organisation and how this in turn supports the long term goals for the organisation.

Making decisions based on architecture rather than technology often means a higher investment cost in the beginning but enables the organisation to reach it’s long term goal. I.e avoiding architecture decisions or making architecture decisions on a case-by-case basis could be efficient in the beginning of these transformations but will over time lead to higher total cost of ownership.




This guidance will support the architecture decisions that are needed for some of the business drivers mentioned above in order to avoid accidental architecture and the long term cost associated with this.

The document builts on CSPA 2.0 but whereas CSPA 2.0 focuses on fulfilling requirements to increase the shareability of services, this document extends the guidance with some of the other commonly occurring business goals. The image below provides a quick overview of how business goals, requirements and features are related.




Implementation of architectural features addresses one or more requirements that in turn contribute to one or more business goals.

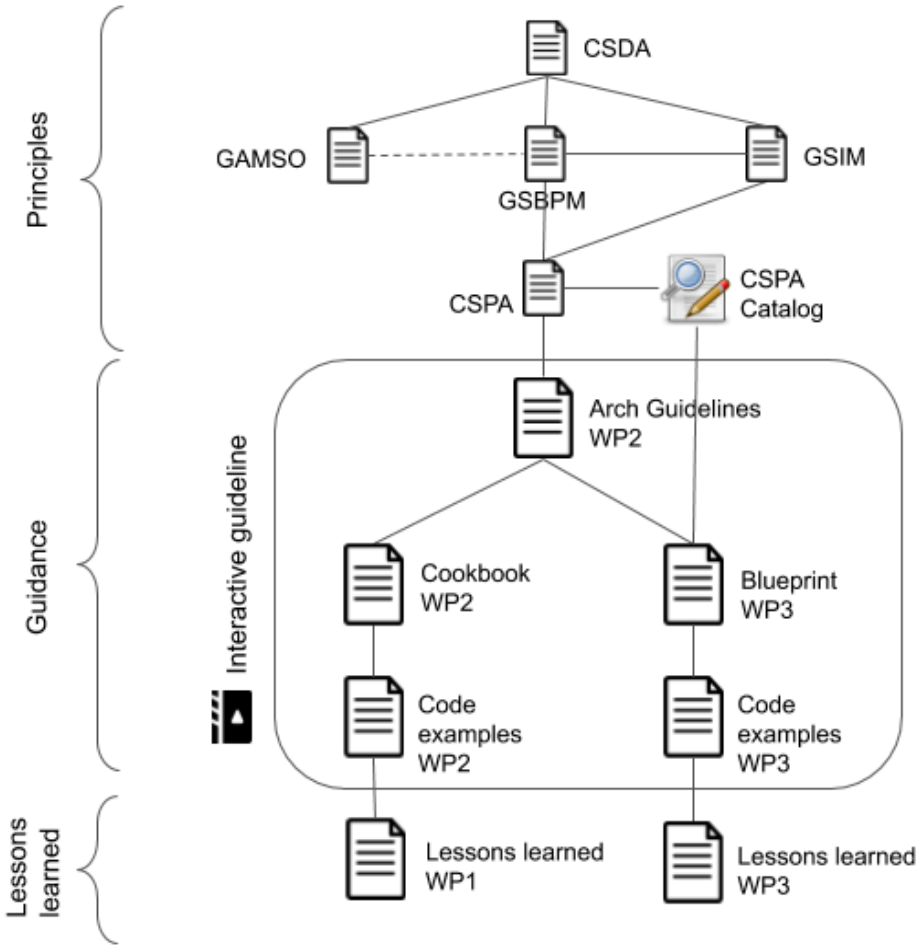
## Target audience for this document

This is a continuation of the CSPA 2.0 Application Architecture and CSPA 2.0 Implementation Guidance. This is also a guide on how to move from legacy architecture to service oriented architecture. A statistical IT-environment would typically consist of both legacy and non legacy systems. By legacy system we mean monolithic. This means that you should have a strategy for how to handle the transformation.

CSPA 2.0 defines 8 different roles that are involved in the application lifecycle management of software for statistical production. This document mainly support the decision making of 4 of them:

- The designer
- The service builder
- The assembler
- The configurer

Most organisations are not organized directly using these roles but instead let existing roles or groups such as CAB:s, it-architects and developers take on the responsibility described by the roles in CSPA. This document is therefore mainly targeted towards architects and developers and is paired together with a more in depth *cookbook-document, code examples* and *an interactive guidance.*



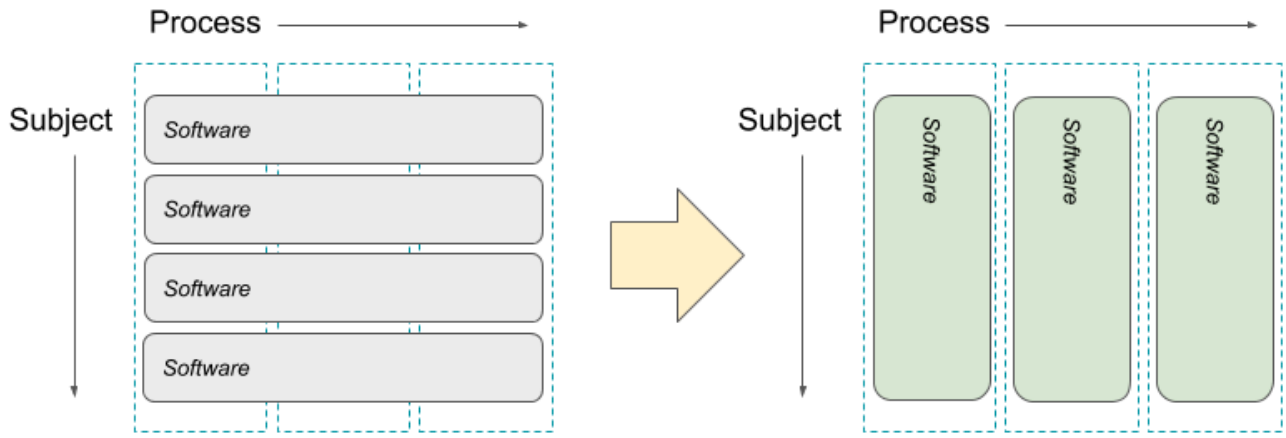

*Overview of how the different artefacts relate*

## From subject matter oriented software to shared process oriented software

It is a strong tradition in many statistical organisations to build and organise their software based on different subject domains such as *Consumer Price Index (CPI), Labour force survey (LFS)* and *National Accounts (NA)*. Some have organised their software to support a single statistical

output/product such as *Quarterly Gross Domestic Product* and in some cases software has been built to support a cluster of statistical outputs/products. This has allowed for a very tailored software targeted specifically towards the unique requirements that these have but this also traditionally leads to a somewhat or very heterogeneous IT-environment that is difficult and expensive to maintain.

Since the 90s the concept of *business process orientation* has had a large impact on many organisations. Within the statistical community, the *Generic Statistical Business Process Model (GSBPM)* has been created and implemented by many statistical organisations. One part of *implementation* is to focus more on the process perspective rather than the statistical domain when organising software to support statistical production. New software is then created and tailored to support processes such as *Error Localisation* and *Applying disclosure control* that can be used for many or all statistical domains (where relevant).




*Transitioning from viewing and organising software based on Subject-perspective to Process-perspective*

A software created for a specific process rather than a statistical domain needs to be configurable in order to meet the specific requirements that each statistical domain has. For the example of *Error Localisation*, this could be the set of validation rules that should be used for a specific statistical domain.

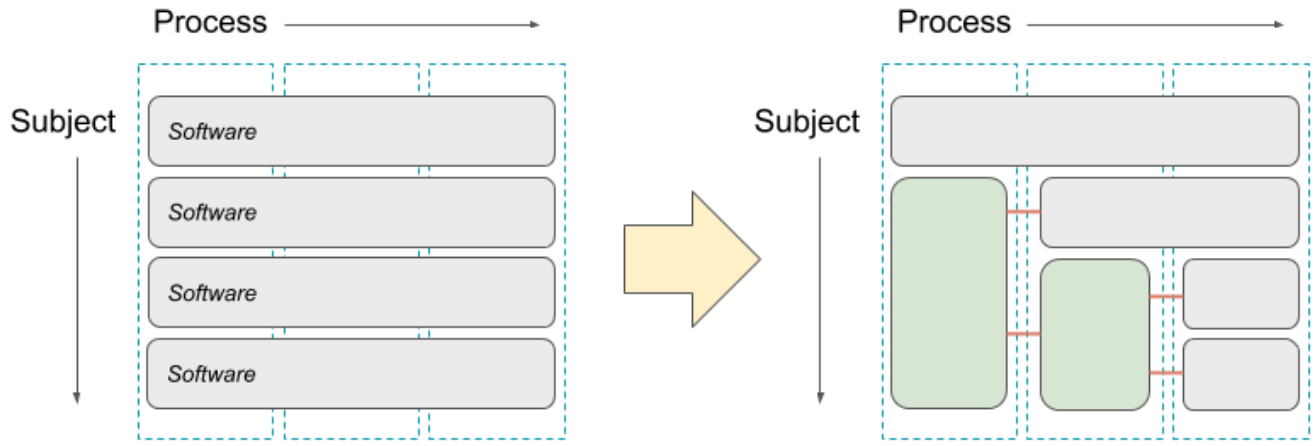
A traditional way of developing software for a specific process is to begin with the requirements for a limited set of statistical domains and then adding more configurability for each new set of statistical domains until the software can support more or less all statistical domains. This transition can be seen as going from *single-use software* to *Sharing and Reusing software* within one statistical organisation.

The high level goal in this transition is often to reduce the amount of code/software that performs similar types of logic which in turn allows the organisation to make quicker business changes and to reduce the total cost of ownership for the software. However, this new software

to business alignment introduces several new requirements significant for the architecture. Some examples:

- Integration - Explicit boundaries between the software supporting the statistical production as a whole, introduces integration needs. Distributed autonomous services often implies decentralized data that has to be exchanged in an efficient way.
- Security - Subject matter / product specific solutions masters its own data created within several process steps is therefore able to enforce security and access restrictions globally. Process oriented services need security solutions that support the fact that they handle data belonging to many different information owners / readers.
- Performance and availability - The workload and data volumes can differ considerably between the process oriented services and the product specific solutions. The chosen architecture must address this fact to live up to the performance and availability requirements.

This transition however takes a lot of time for most organisations since it involves both investments to be made to the entire software portfolio as well as changing the perspective for the staff involved in both developing and using the software. For many organisations, the target could also be set to less than a full transition.




## Agile business processes

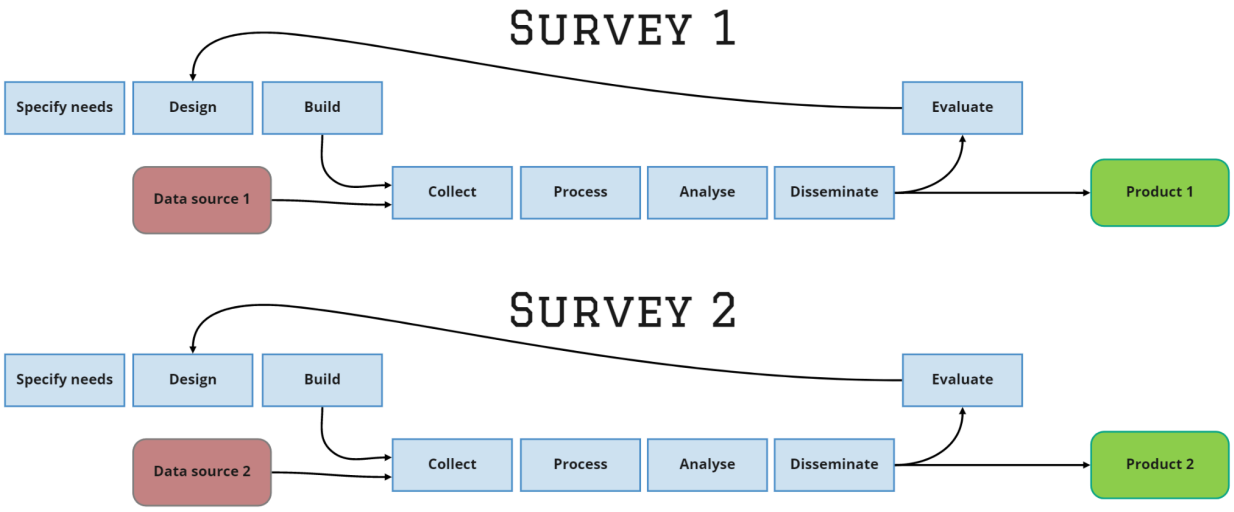
Most statistical organisations are looking into utilising the effects of the ever increasing digital transformation of society. The availability of new digital data sources as well as the expectation to respond quickly to demands for new types of statistics require that the statistical organisation can quickly respond to changes. New channels for data collection and for dissemination often require investments into new or updated software. The transition from *subject matter oriented software to shared process oriented software* mentioned above would enable this investment to

benefit multiple statistical domains since the software created to support data collection from a specific channel can be re-used by multiple surveys.

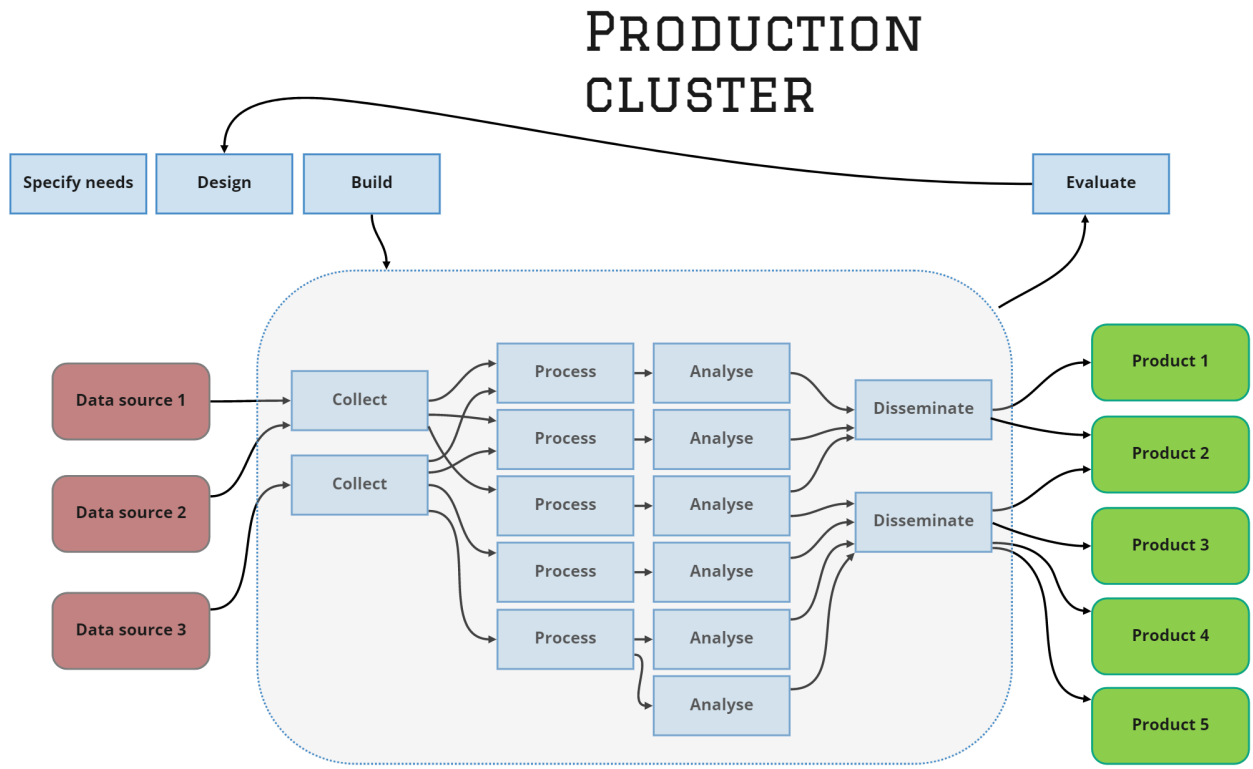
Mixed-mode data collection adds a *many-to-one*-type of requirement when it comes to data integration. However, some new data sources have the potential to support or even replace multiple existing surveys. In these cases, the data collected from the new data source should be accessible for multiple subject matters *Process* and *Analyse* process phases. From a data integration perspective, this leads to a *many-to-many*-type of data integration.

To respond more quickly to demands for new types of statistics, many statistical organisations aim to ensure that the existing data can be shared and re-used. Data that has been collected and processed can through further processing and analysis be used to create new statistical products in order to meet the requirements for new insights more quickly.

The shift in viewpoint here means that the GSBPM-phases cannot be seen as a set of *one-to-one*-type of interaction but rather that a design must be set up for a complete *production cluster*. To support these types of agile business processes, the data integration architecture must support more complex routing schemas.



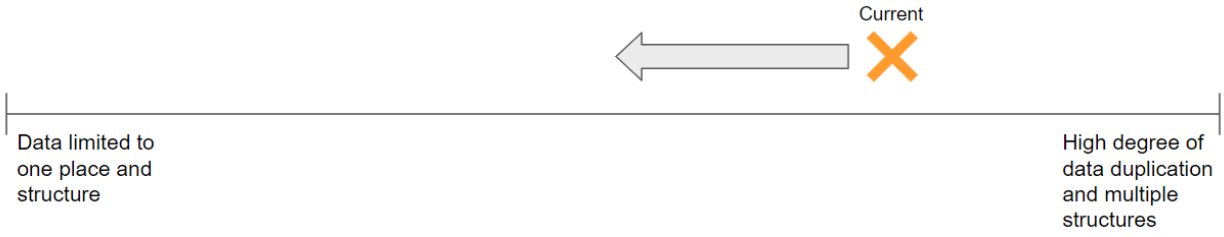

Example of a set of simple surveys that follows a set of *one-to-one*-data integration patterns




Example of a more agile business process that requires a more advanced data integration architecture

## Enterprise data management

Many statistical organisations implement strategies for better data management on enterprise level. Following the motto of treating data as a high value asset, policies and architecture to support better master data management and going towards more homogeneous data structures is often implemented. Limiting duplicates of data and harmonizing the data structures used to hold the data can make integration between IT-systems easier but going too far towards this could also lead to unwanted effects such as difficulties in updating IT-systems and risks of introducing *single point of failures.*




Targeting different points in this spectrum will have consequences on the architecture and how data integration can be structured but also how much effort that goes into synchronizing

different development initiatives since a more connected data architecture also requires a more synchronized development effort.

## Reuse and sharing services

There are multiple challenges involved with reusing and sharing services. Statistical organisations are different in terms of size, organisation, culture, history, etc. Some other differences, such as technical platforms, can be addressed by architectural decisions. There are solutions such as containerization available to improve the level of portability of a service.

There are also other things that could be shared beyond services. Reference architectures, algorithms, information models and test data are all examples of things that could also be useful related to services but the focus in this document is to make sharing of services as practical as possible.

A service could be shared and reused in different forms which each has its benefits and drawbacks.
Different ways of reuse:

- Code / libraries are shared. Here the *build, deploy and consume* parts have to be managed by the reusing organisation.
- Deployable binaries / packages are shared. Here the *deploy and consume* parts have to be managed by the reusing organisation
- “Centrally” deployed runnable services are shared. Here only the *consume* part has to be managed by the reusing organisation.

Due to the increased interoperability that technologies such as containerization provides, it’s quite common today for open source projects to combine sharing code/libraries together with deployable binaries. This lowers the complexity of reuse since sandboxing techniques can be used but this also puts requirements on the autonomy of the service. A service that is shared but also requires a combination of multiple other services to be able to operate will be more difficult to reuse unless the full set of services required can be bundled as a deployable package.

A service could also be shared and reused in different contexts. A first version of sharing and reusing is often sharing and reusing a service within the organisation or a smaller community of organisations. This allows for building incremental knowledge about the needed aspects of shareability but could also mean that a lot of time has to be spent on re-engineering since the design and architecture for the service might need to be changed multiple times.

## From legacy technology to SOA and/or cloud

Since technology advances in such a rapid way, a lot of organisations will at some point end up in a situation where the technological dept has to be managed in some form. If the technological dept becomes big enough it might not be feasible to update all legacy technology at once. In some cases the shift towards more modern technology is also bundled together with other strategic initiatives such as the business drivers mentioned above. This could make the transition challenging and requires a plan that also includes the architectural aspects of the transition.

One of the things that can make the transition from legacy technology to modern technology very complex is the integration between it-systems. Some services can have a lot of consumers which means that updating the technology for these might require all the consuming services to be updated. In these situations, the dependency between services is not only on a technological level since the development and governance of these services has to be coordinated.

A more complex situation like this could therefore make the cost of transitioning to newer technology even more expensive or risky and this in turn can lead to further delaying the transition which increases the technological dept even further. Keeping an updated architectural view of dependencies between services is therefore an important basis for investment decisions related to services.

Mitigating solutions in complex situations can be to run multiple versions in parallel of some services but since this in turn could make service maintenance, data management, security and performance more complicated, it should only be seen as a temporary solution. Another important perspective of SOA is to ensure well defined contracts and interfaces between services which could also simplify the governance of services.

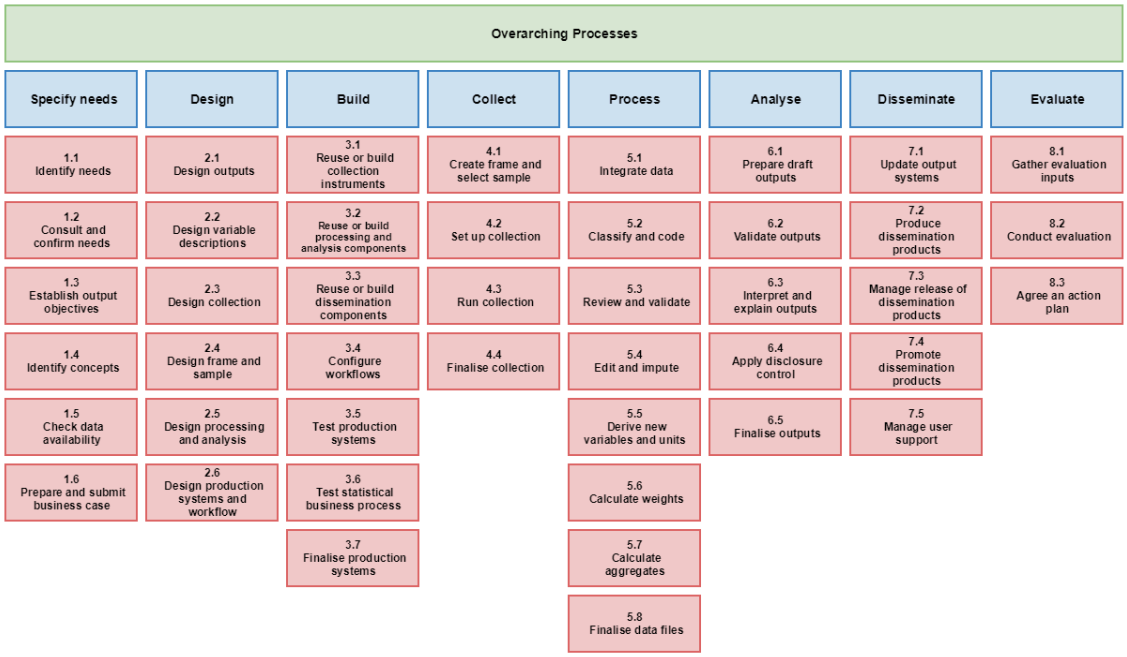
Even though technology can often be used to bridge architectural paradigms such as SOA and more recently cloud technology, it’s not often a solid long term strategy to use technology for this. Even though old monoliths could be converted to new SOA-based monoliths, the recommendation is almost always to reengineer the solutions and to break down large monoliths into smaller services that communicate using well defined interfaces. The same type of recommendations can now be seen when moving towards cloud technology. The old services can in many cases be packaged in a way that makes it executable within a cloud environment but without reengineering, the services will most likely not perform very well. This approach, often called *lift and shift*, tends to lead to a higher cost of running the services.

# Concepts and context - “What”

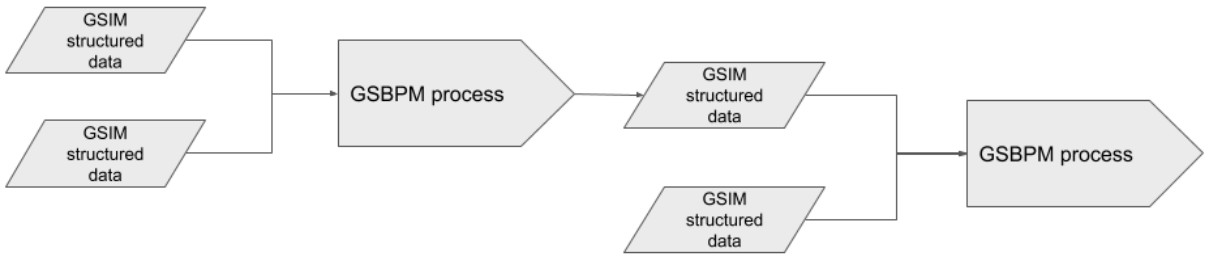
## Metadata driven, GSBPM

Generic systems must in some way be metadata driven. Metadata can take its form in different scenarios from metadata about a statistical design or a calculation design, variable design like classifications and codelists to system deployment configurations. Different types of metadata are required to achieve different types of automation and shareability. The notion of design first and then consuming the design in the production process is key.

GSBPM should not be interpreted as a linear model. The flow of information in the GSBPM often involves processes that are iterative, repetitive and parallel. For example Collection and Processing can be done in parallel.



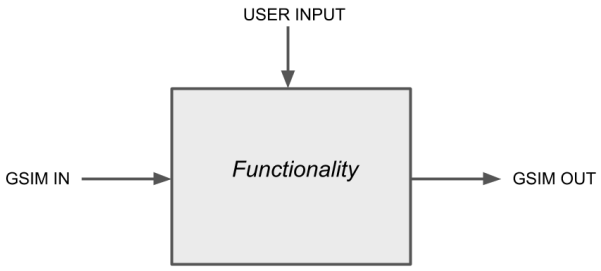

Statistical production from a GSIM-perspective could be seen as a series/network of different types of data creating/processing/manipulation steps. Each step requires different types of GSIM-objects and the results from the step are GSIM-objects that in turn are used by other steps. As shown in the image below we can see how a GSBPM process consumes GSIM structured objects and produces GSIM structured objects.




Conceptual process diagram.

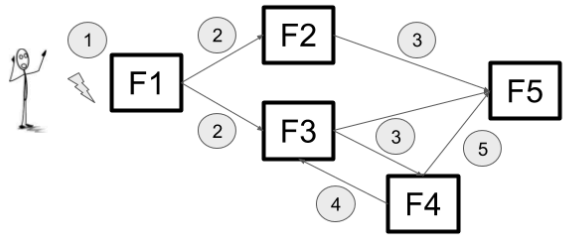
Different names exist for the lower levels of work in statistical production. For steps closer to the GSBPM-phaze, it will in many cases be a specific method or algorithm that should be performed but for other GSBPM-phazes it could be described as a *sub-process*, *an activity* or a *task*. In this document we use the word *functionality* to describe the conceptual level of these types of steps.

For most process oriented *functionalities,* the connection to GSIM is described as following: GSIM input -> *Functionality* -> GSIM output




Some *functionality* can be automated while others involve user interaction. What triggers a *functionality* can be that some GSIM-information becomes available. For example - the *functionality* of *error localization* can be performed automatically after a required *Dataset* is available. But can also be triggered when a user decides to perform the *functionality.*

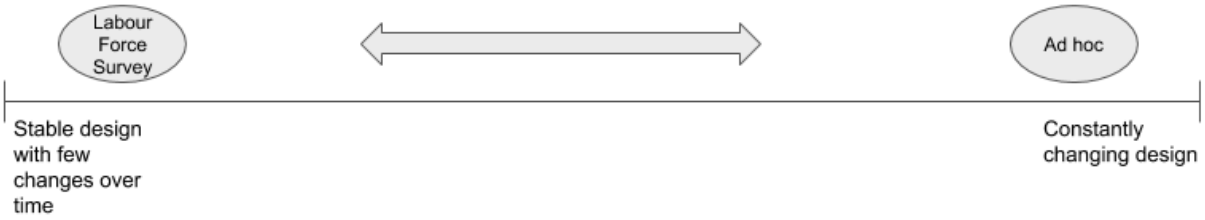
Business functions are accomplished by chaining *functionalities* that exchange GSIM-objects




An example of a Business Function as a set of *functionalities* chained together in a design. In the image above we can see how functionalities are chained in a specific execution order that

begins with a manual action and some of the functionalities are executed in parallel as shown by the numbers in the image above.This can be defined as design metadata for a calculationround.

Statistical programs have different levels of stability in their designs.



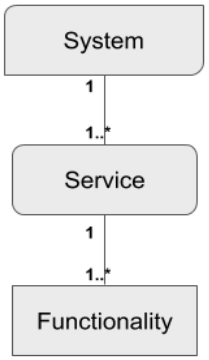

For example, the image above shows how the labour force survey tends to have a more stable design vs ad hoc surveys have a more volatile design. If a statistical program is on the right of the scale and has a more volatile design it's more important to have metadata about the design separate from the logic.

Beyond *functionalities* related to GSBPM-processes other types of *functionalities* are needed to support the production process. Examples of other commonly required *functionalities* can be *functionalities* for data storage, graphical user interaction and *functionalities* for security.

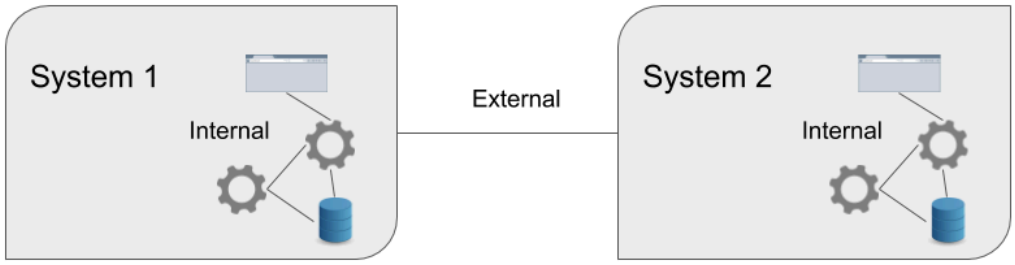
In order to go from the conceptual level of *functionality* to something technical we need to implement some form of technical implementation. From the CSPA-standard we have a relatively broad definition of *Service* but later in the document we describe how this document makes use of words such as *Systems, Services, Applications, Functions, Code,* etc and that they can be seen as different forms or different levels of technical implementations for a *Functionality.* The sum of the implemented *functionalities* are what's needed to support statistical production.

# System, service and functionality

This document makes use of terms like system, service and functionality. The image below shows the relations between them:




A system can consist of one or several services. A service can consist of one or several functionalities. In CSPA and TOGAF the definition of an Application is - A deployed and operational **IT system** that supports business functions and services; for example, a payroll. Applications use data and are supported by multiple technology components but are distinct from the technology components that support the application as shown in the image below.

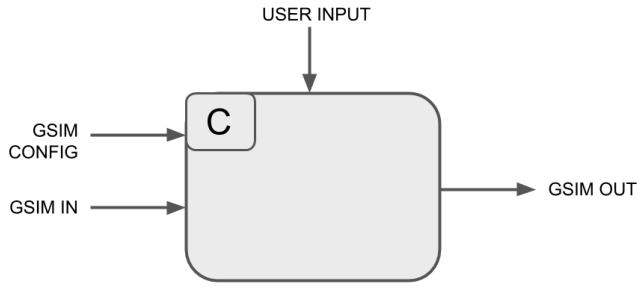



The system border is the ultimate point where integration must follow GSIM for communication of data. The internal structure of a system can be hidden from the outside world, especially for older systems. However, the goal is always for the components to be loosely connected even within a system. That means that statistical services should always communicate GSIM objects.

## Information and services

The definition of services from CSPA for three types of services: Statistical Function service, Entity service and Utility service.  An utility service could be a VTL execution engine. An entity service could be a classification service, and a Statistical function service could be error localization.

A service should have a configuration. It should require GSIM input entities and expose GSIM output entities. A configuration could be context related like Program and program cycle but could also be other types of metadata.

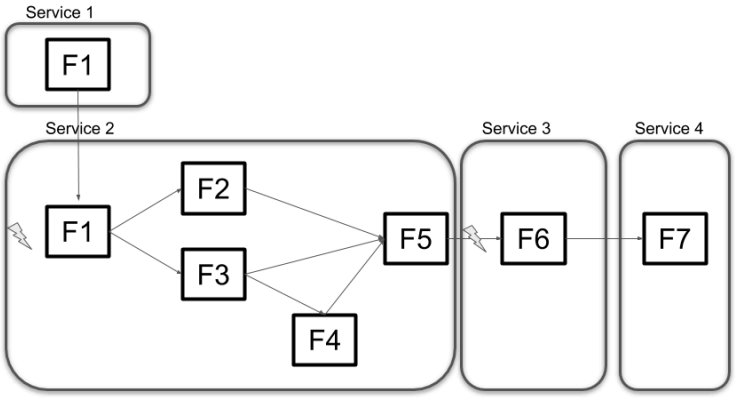



A service often consumes information internally and externally. Some of the information is master information, other is secondary information. The external information structure must be GSIM objects but the internal dont have to be. This is true in many cases for older systems.

Enterprise data management involves both managing the physical storage and transportation of information/data as well as managing the structures used to store and carry data.

## Clustering of functionalities to services

A service can be composed by a cluster of functionalities as shown in the image below. The image shows the logical view of the information flow. The service in this image is the abstract view:

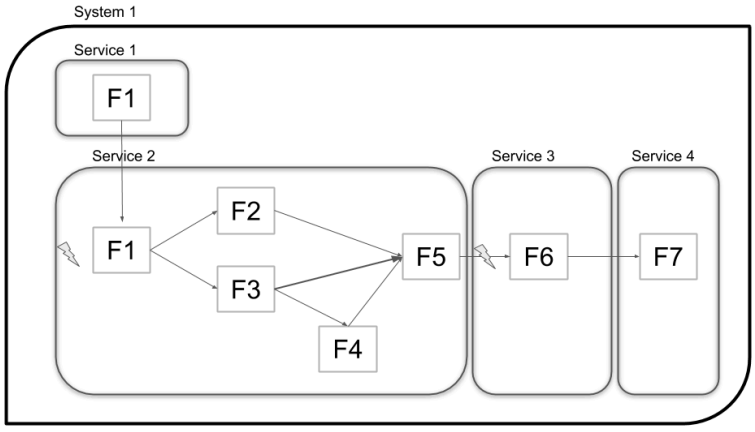



This means that depending on how a system is composed the workflow for the CSPA role configurer can differ depending on the composition of a system. Some functionalities can be encapsulated in services, scripts, and configurations that are dynamically loaded.

Different statistical subject domains/products. Will make use of different combinations of services in order to fulfill their needs. In most cases a service is built explicitly implementing the *functionalities* that should be part of the service. There are however cases where the service is built in a way that allows for dynamically loading/configuring the *functionalities* that should be used.

## Clustering of services to systems

A system can be composed of multiple services. Systems can also consist of UI, storage and integration points.

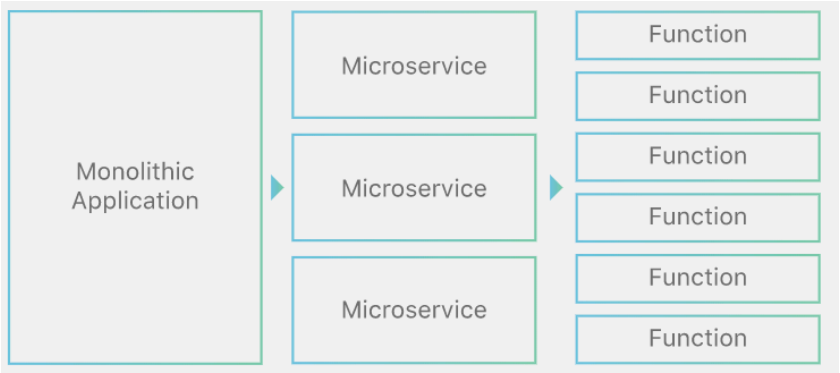


## Implementing functionality

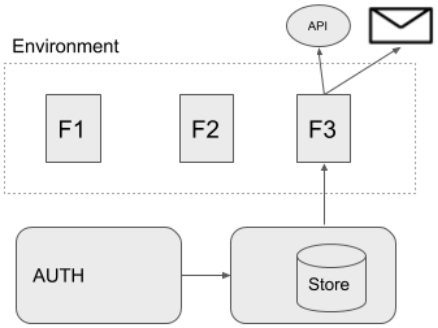
Functionality can be implemented for use in runtime or in compile time. A functionality must be implemented in some technical framework(.Net core, R, Neo4J). A functionality will also be encapsulated in different architectural styles like Ports and adaptors also known as Hexagonal Architecture. For more information on Ports and adaptors please see the CSPA document.

### Function as a service

When thinking about how the functions and services should be implemented some considerations should be made on how the service is going to be deployed. Function as a service means that you can define a function without provisioning an entire system.



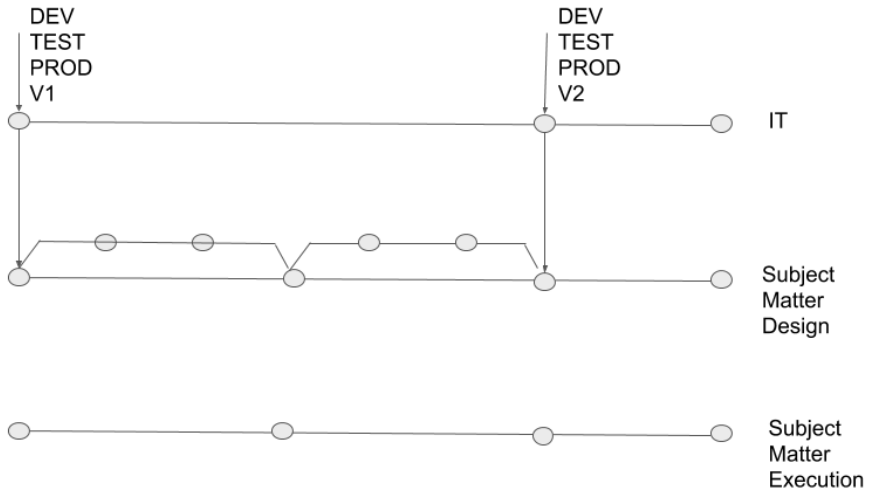

The implementation can vary depending on where it is deployed and functions can be written in different languages and hosted in different environments.




In the image above functions are deployed as code. Function 3 pulls data from a datastore service and when it is finished it calls a rest api and a mail service.

## System context, environments and versioning

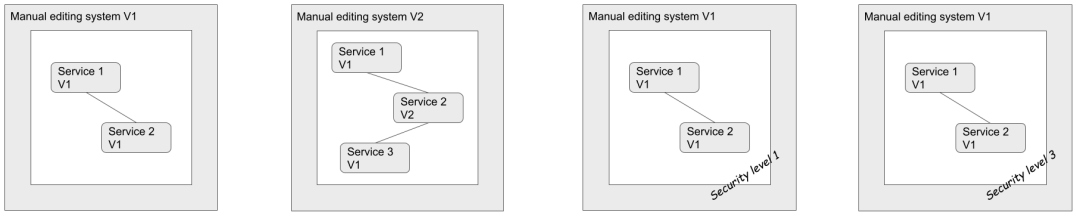
Systems can be used within different contexts. A common context is the ALM-perspective, development, test and production. All contexts may require different environments. Within the production environment a system can also exist in different business contexts like different statistical programs or statistical cycles. This context can be deployed programmatically or in runtime.




Different environments could be created for production to separate the business contexts. The purpose for creating different environments could be due to performance, security, easy versioning of systems, keeping production environments sanitized with regards to old data/information etc. The subject matter design also has a version life cycle that is different from the versioning of the IT-Service. For example a calculation round can be composed of many calculations. Changing that design should not require a new deployment of the IT-System. The design of that calculation round can then be executed in several instances. This is why it's a good practise to extract the subject matter design from the core logic. This also means that IT personnel do not have to be involved in the statistical design. The subject matter design is a part of the production environment. The production environment should enable working with the subject matter design between exekution of the design. The release of new system versions should not affect the subject matter design. If this is the case the subject matter design must be tested where applicable in the IT-test environment. Another way to partition and execute the design is to use the environment. Here are some different approaches to setting up environments/systems:

- Option 1 - All Statistical Programs run in the same environments - the same system instance is used by all Statistical Programs
- Option 2 - Manually creating different environments for different Statistical Programs or Statistical Program Cycles
- Option 3 - Automatically generating environments or instances of systems based when a new Statistical Program or statistical program cycle is created
- Option 4 - Continuous integration/Continuous Deployment or deploying functions

This means that a system can exist in different versions in different production environments.

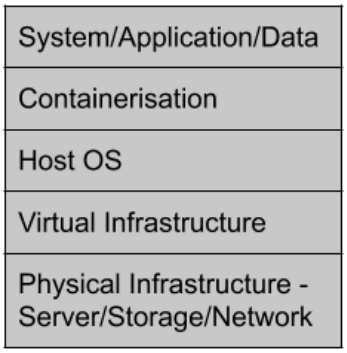



In the image above we can see how the same system called manual editing is deployed in different versions with different services in different security levels.

# Architectural features for business drivers - “How”

An architectural feature is a high level solution for addressing one or many requirements. The target requirements for this document are derived from the business drivers above. The term feature is reused from CSPA 2.0 Application Architecture, which also includes many of the features stated here. A feature contributes to one or more of the requirements that often exists in statistical organisations and can be implemented by various solutions and technologies.In this document we target a subset of the CSPA Features that we deem most important for sharing or have some ties to the other work packages..

A feature could be implemented in many ways and in different layers. Some solutions could be built into the services, while others could be implemented by the service consumer.  The purpose of this chapter is to try to give input to the process of making architectural decisions.



## Sandboxing for exploration

A sharable service should be able to be sandboxed. This can be done in several ways. One way is that the service is packaged in a way that allows statistical organisations to explore the service in their environment. This means that the package could generate databases and runtime components automatically. Other ways to sandbox the services are to make the service runnable in an open environment so that statistical organisations don't even need to download the service to explore its functionality. Sandboxing also offers the possibility of managing security in other levels than the application level. A strategy could be to handle the security on the sandbox endpoints.

## Security

The first step is to understand your current statistical environment. Statistical organisations today have complex environments. Remote work, globalization, and cloud computing have dramatically improved efficiencies and productivity in the workplace – but these changes have also added new vectors for attackers.

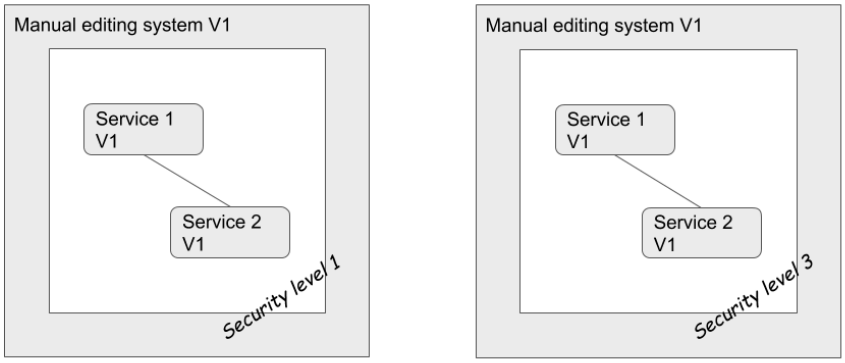
It’s important for statistical organisations to have control of data that the government body deems to be sensitive information that must be protected. The IT-systems must therefore reflect this by taking steps to protect the information.

Layered security, means protecting digital assets with several layers of security. The concept behind layered security is simple. If a hacker manages to breach one security measure, all sensitive data is still protected by the other layers of security that are in place. This makes it harder for a hacker to perform a successful cyber attack. In this layered approach, each layer of security can work together to ensure enhanced protection against threats. Each layer can implement authentication methods to ensure that a minimal level of trust is necessary for the solution.

When using sandboxing environments like containers it is important to consider encryption of data "at rest" based on classification levels, and at least encryption with the key provided by the service provider. It is just as important to ensure that configuration and secrets outside the container are safe and encrypted.

Security can be implemented in many ways in different layers and it’s important to have a clear strategy on how to handle security in different layers and also in the information layer for example to have classification levels on the information contained by the system. Not having this from the start will drive the costs and decrease shareability.

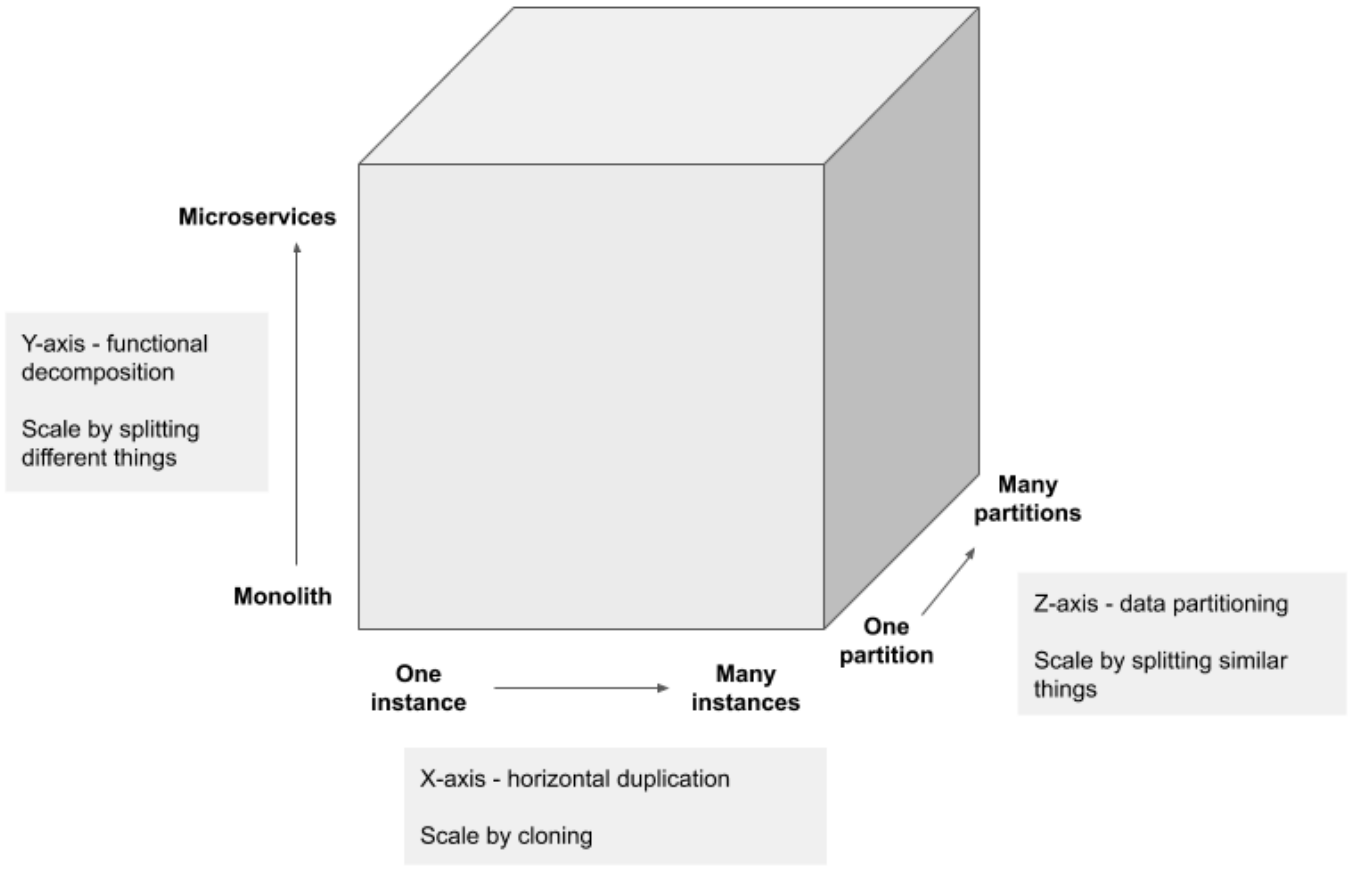
For older silo systems, security can also be handled in different ways. In the image below we can see how the same system is deployed in different environments depending on the classification of the information that the system handles.




Security must be configurable. The role of automated configuration management is to maintain systems in a desired state in order to reduce cost, complexity and errors. This is especially important in a Zero Trust architecture.

## Scaling

Scaling is the process of making software more capable of handling increased capacity and availability. A common way of defining different scalability solutions is the three dimensional cube model:




X-axis scaling load balances requests across multiple identical instances of a system in order to distribute work.

Y-axis scaling is about decomposing a system into multiple different services where each service is responsible for a group of closely related functions.

Z-axis scaling is often combined with X-axis scaling and adds the capability to make each instance of a system responsible for a subset of the data based on some routing rule. This approach is commonly used when scaling databases.

Considerations driven by business drivers:

| | **X** | **Y** | **Z** |
|---|---|---|---|
| **From subject oriented software to process oriented software** | Increased workload and data volumes will require scaling capabilities. X-axis as well as Z-axis scaling will be fundamental to fulfill the availability and performance requirements. | Very beneficial solution when process orienting the software. Functions supporting different parts of the processes are supported by different services. The granularity of the services will differ but still be supporting a part of a process instead of being hard coded to a specific subject. | *See X.*<br><br>Consider the current context to be used for routing rules. For example, the data persistence layer could be partitioned in alignment with the different Statistical Programs. |
| **Enterprise data management** | Not specifically, but X-axis scaling is almost always applicable. | Storage solutions with high data volumes and increased number of consumers needs extensive scaling. | |
| **Reuse and sharing services** | Workload and data volumes will differ among potential consumers of a service. Scaling solutions are important to support. | Services with limited functionality are easily shared. Y-axis scaling is inferred. | *See X.* |
| **From legacy technology to SOA and/or cloud** | Service orientation principles such as loose coupling and standardized service contract simplifies the implementation of all of these scaling solutions.<br><br>Scaling solutions like these are fundamental promises of all cloud computing platforms. | | |

## Resilient services and error handling

For a service to be attractive and reusable it must implement a decent level of fault tolerance and recoverability.

In the event of an interruption, temporal high latency or other failure it can to some degree still be possible for the service to operate as intended. Failure due to underlying blocking calls could be handled by solutions such as the *Circuit Breaker pattern* to effectively prevent running out of critical resources such as memory and threads.

Failure due internal faults must always be handled and logged to enable troubleshooting, correction of errors and other improvements. To increase the reusability of the service, the logging should be created in a way that doesn't depend on what type of logging frameworks that are available in the environment that the service is running in. This could either be done by using a *Sidecar*-pattern or by leaving the responsibility of handling logging to the environment in which the service is running - i.e. relying on the layer below to handle logging.

For enabling status monitoring of services, a health check system can be implemented. Since the service itself can end up in a state that makes it non-responsive, a simple health check should be set up in the environment in which the service is executed.

Recoverability can be defined as the degree of which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system. This is a very important requirement to live up to in order to improve the end-user experience as well as reducing IT-support costs. Concepts such as statelessness, backup and failover solutions can be adopted.

Within the development of services, *chaos engineering* can be used to efficiently improve the level of resiliency. This often implies the use of specific external tools for creating unexpected conditions within the runtime environment of the services, in order to identify possible weaknesses.

## Versioning

Versioning is always a concern in an architecture. Versioning is also a CSPA feature and is crucial for sharing. One thing to consider is that there are different things that can be versioned like code, binaries, endpoints and metadata. In the What section of the document we emphasized the importance to have a holistic approach of the hole chain regarding versioning. In this chapter we will look at some key aspects of versioning in different contexts:

## Service and code versioning

When working with code and release management it’s important to have a clear structure for how versioning is handled:

[Major release].[Minor release].[Revision/Bugfix)]

Major means major changes with new functionality. Minors are minor functionality changes that are also backward compatible. Then comes the bugfix version which indicates that the version is bugfixed.

<Assembly: AssemblyVersion("1.2.1.0")>

Example: 1.2.1.0 = Majorversion=1, Minor version=2, Bugfix=1,For komp=0

This will ensure reuse of services and code.

## Endpoint versioning

Integration of endpoints refers to the collection of functions, messages, views, information objects, etc. that are exposed to one or more consumers to solve a specific task. An integration point can be implemented with different technologies, e.g. Web services, Web API:s, database objects, message management. It’s always got to plan and implement version control for an integration point, even when the integration point only has a single consumer. This is because version management has an effect in the management phase. Version of system and version of integration point are different things. The system can come in many versions between each version of the integration point. It is mainly when backward compatibility is broken that the integration point comes in a new version number. When the integration point has changed so much that the backward compatibility is broken, so-called "Breaking changes", a new version of the integration point must be commissioned. As it’s consumed by other consumers, it must be put into operation in parallel with the previous version. The two versions must then coexist until the last consumer has upgraded.

## GSIM structure versioning

Metadata services will need to be version managed. It must be possible to refer to and access a specific version of an object in time.

Here are some principles för versioning metadata:

- Metadata should be append-only. No objects should be removed. They must exist for history even if they are incorrect. However, they may have a status that can be filtered out..

- It should be possible to find all versions of an object and see in what order they come.

- From one version it should always be possible to find relations to all other versions.

- It should not be possible to change a fixed version (Immutable). It includes references to other objects. If an object refers to another object of which there will be a new version, it must not be possible to change the reference to the new object.

- An object cannot have a status determined if its sub-object (object it refers to) does not have a status determined. For example, a variable cannot be determined if it points to a concept that has not been determined

- It must be possible to work on a new unspecified version without affecting production

- It must be possible to access the most recently determined version of an item, regardless of whether there is a later undefined version or not.

### Information object instance versioning

Value quantities are used in the production of statistics. These consist of pointers to specific versions of Values. These Values are found in one or more Code Lists and Classifications. Value sets must also be version managed. To use a version of a Value Set in production, it must be put in context .

## Containerization

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. How containers are composed depends on what the goal is to achieve with the containerization.

Containerization is a very fundamental feature for achieving sharing and reuse of statistical services and is further described within WP3 of the IS3 project.

# Integration patterns

## Patterns for integration of services

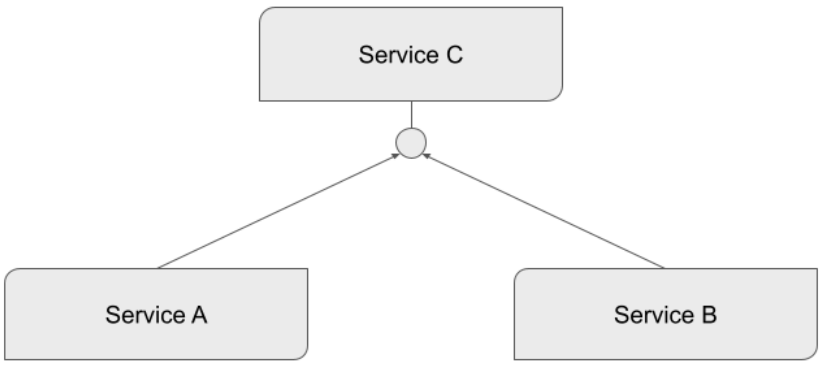
Appropriate integration patterns will always be in use when building software solutions composed of multiple services. Each information flow between the services needs to be carefully implemented based on a set of architectural significant requirements. The specific requirements in different parts of a larger solution often lead to an intentional mix of different patterns.

In this document three main basic patterns will be discussed:

- Point-to-point
- Orchestration
- Event-driven

### Point-to-point

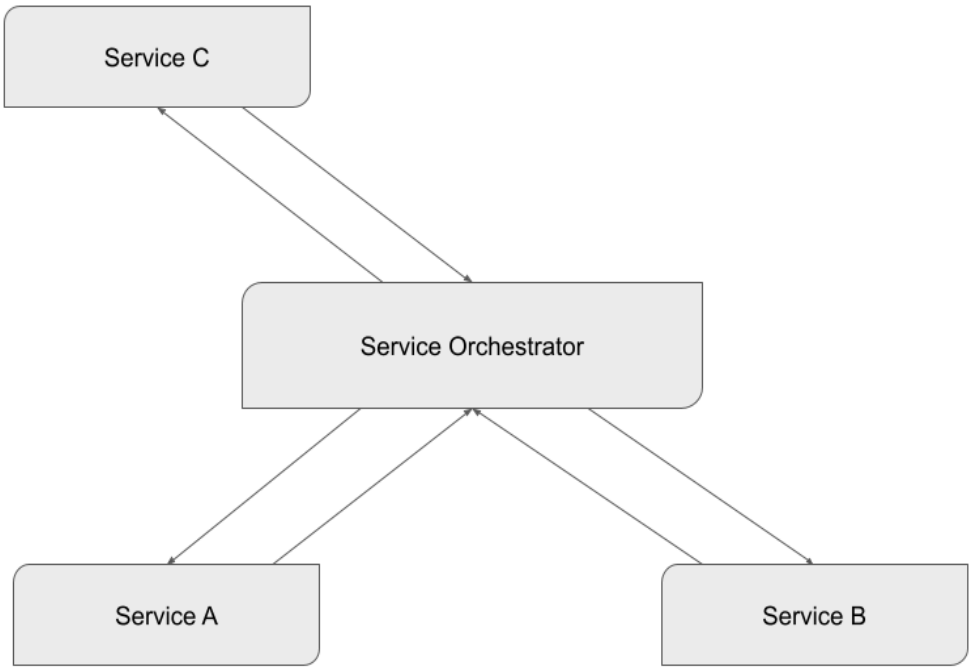
A service consumer knows about the service and is able to directly execute requests against it. A common example is request-response communication against an HTTP Web API.




Service A and Service B consume Service C. They make requests against a known location of Service C and their respective response is expected to arrive in a timely fashion.

### Orchestration

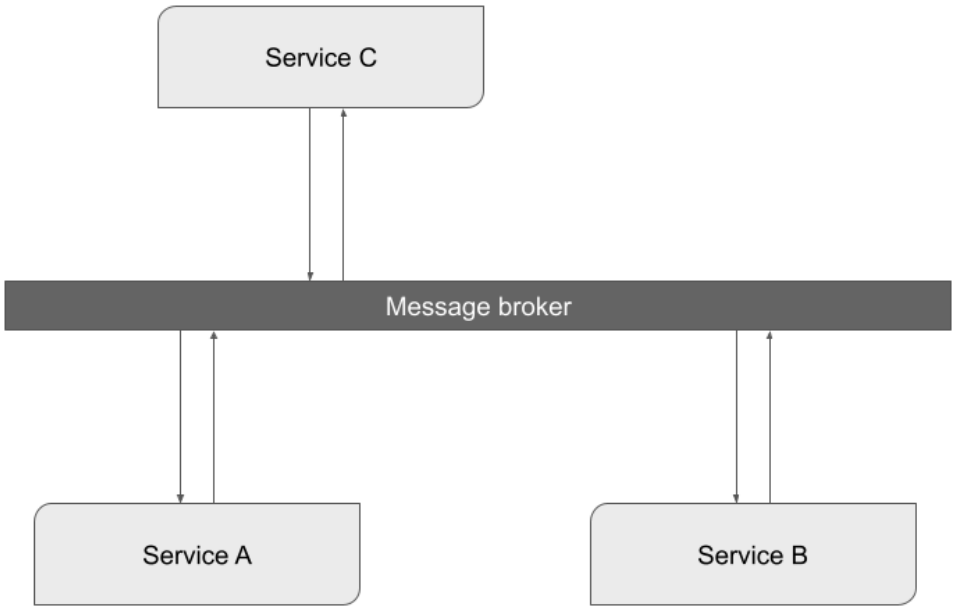
A central service coordinates the interaction between the services. Invocation, combination and cross-service transactions are centrally managed.




The central orchestrator with its own runtime is in charge of all interaction and implements business logic such as decision making and workflows. The participating other services do not know about the internal workflow of the orchestration.

### Event-driven

A message-based pattern where events are published to a central message broker. Each participating service subscribes to events from the broker and knows exactly how to handle them in its own logic.




The services participate in the overall solution by publishing and subscribing to messages. Decision making is decentralized to each service.

### General Comparison

Interaction:

| | **Point-to-point** | **Orchestration** | **Event-driven** |
|---|---|---|---|
| **Service interaction cardinality** | One-to-one | One-to-many | One-to-many |
| **Synchronous / Asynchronous** | Typically synchronous request-response but asynchronous could also be implemented. | Both | Asynchronous |
| **Additional middleware needed** | Not for synchronous interaction. | Orchestrator engine is often needed. | Message broker needed. |

Pros and cons:

| | **Point-to-point** | **Orchestration** | **Event-driven** |
|---|---|---|---|
| **Pros** | ● Low complexity offers low implementation cost. | ● Centralized governance. | ● Reliable information flows.<br>● High agility.<br>● Low coupling.<br>● Extendable.<br>● Scalable. |
| **Cons** | ● As the number of connected services grows, the effort of keeping up the overall availability and maintainability could increase exponentially.<br>● Hard to adhere to the principle of service autonomy. | ● High complexity..<br>● Low agility.<br>● Single point of failure could be introduced.<br>● Risk of putting business logic in the wrong place creating architectural dept. | ● High complexity.<br>● A new mindset may require new skills.<br>● Requires middleware such as a message broker. |
| **Mitigation actions** | ● Standardize service contracts.<br>● Standardize the client software | ● Use it with clear intent.<br>● Be careful of vendor specific functionality. | ● Train people.<br>● Architectural governance on the overall solution. |

## Choice of pattern in a statistical production context

| | **Point-to-point** | **Orchestration** | **Event-driven** |
|---|---|---|---|
| **Replicated data**<br><br>**(see separate chapter below)** | Point-to-point has some drawbacks here because of its typical synchronous style. For data to be replicated it is often necessary to implement chained service calls that could be very brittle in case of partial failure. | Distribution of data across multiple services is manageable through the orchestrations even though the pattern in itself does not encourage this. | The event-driven pattern is very suitable here Message based interaction creates a solid foundation for reliable eventual consistency across multiple services. |
| **Storage vs integration** | The concept of a “shared database” is not considered a relevant pattern when integrating shared statistical services. Autonomous services need to own their own storage which is considered strictly private. However, the point-to-point pattern, with its pros and cons, could be a more service oriented way of the same basic idea of requesting known resources. | Storage is private to the services and is not a part of a solution for integration. | |

| | | | |
|---|---|---|---|
| **Execution of process steps, parallelism and timing** | Given that process steps often are implemented in different services the point-to-point pattern does not really fit in here. Process control needs something external that keeps track of the process steps. | Orchestrations can in a natural way implement this. | Smart services that act on subscribed events is a suitable pattern for achieving flexible process step execution. Parallelism and timing of tasks independent of each other are very feasible with this pattern.<br><br>A good example of this could be when the data collection process is designed to immediately start manual editing for received data when respondent to a questionnaire submits. Services for error signaling and manual editing could be taught to subscribe to events specific for this scenario and then execute their logic in an event-driven manner. |
| **Service sharing and reuse** | The low threshold and simplicity of the point-to-point pattern is appealing when getting started with sharing and reusing services. | Greater complexity than point-to-point. | Greater complexity than point-to-point. It can also be challenging to describe the service interface. Events expected to be handled and published from the service need to be clearly defined. |

| | | | |
|---|---|---|---|
| **From subject matter- to process oriented software** | When tearing down old subject matter oriented software and likely replacing parts of it with new process oriented software we get integration needs between them.<br>Point-to-point will be applicable but it also comes with some drawbacks. See right columns. | | Process oriented software implies the fact that a large number of consumers may exist. This was not the case in the subject oriented world. To support reliable information flows from the process oriented services to many consumers a message based pattern is very rewarding. |

### Data only once vs data duplicated/replicated in services

When multiple services interact together it is often crucial to be able to keep local replicas of data within the services to maintain availability and low latency. Depending on several characteristics of the data this can be more or less appropriate. Factors that may constrain the level of distributed copies of data:

- Security: Access to sensitive data is easier to control when limiting the number of distributed copies. Policies can be applied close to the data.
- Storage: Large data(sets) can be challenging to store in multiple places.
- Consistency: Copies of data can be difficult to keep in sync with the single source of truth.
- Enforcement of archiving and disposal rules.

Content metadata is an example of data that is particularly suitable for replicating whereas the entire content of a base register is not.

### Versioning solutions and integration patterns

The need for information exchange between services becomes even more complex when versioning is considered. Different versioning solutions (see above) will influence the design of the integration.
For example when running multiple service versions, each in its own runnable instance. The different instances may more easily interact with each other when using a message based pattern. Of course, solutions exist for every combination of versioning solution and integration pattern, but some may be more preferred than others depending on specific priorities.

### External platforms and integration patterns

When introducing larger external platforms (for example computing- or process control platforms) into a statistical production environment, the choice of integration pattern is affected. The in- and outgoing “interfaces” from a platform often infers specific patterns. As stated earlier, a mix of different patterns is natural.

## Documentation, Multilingual support & Open source

These three features are key features for enabling sharing. Services must be documented at least in English in addition to the local language(s) of the organization developing the CSPA Service. It is highly recommended that organizations that have made translations of the documentation of a CSPA Services in additional languages make them available to the community. If the service includes manual operations, it also must be possible to change the language of the graphical user interface. Code and comments should be in English. CSPA Services should also be able to support input and output in multiple languages where applicable. CSPA Services should be distributed as open source with an appropriate license approved by the Open Source Initiative, depending on organizational policy. This provides transparency and visibility and promotes collaboration, easier distribution, and the ability to create automated deploy pipelines. The use of Open Source is also now a central part of many statistical organizations' portfolios, and hence the adoption of Open Source components should be more accepted.