Generic System-of-Systems Description (GSoSD)

The Arrowhead Core Systems

**Abstract**

This document outlines the 5th generation of the Arrowhead Core systems, which offer functionality expected to be needed for the majority of use cases where Arrowhead is applied. It contains a high-level architectural description of what problems the Core systems solve and how they interact.

The primary purpose of the document is to present the functionality offered by the Arrowhead Core systems. Its architectural aspects are defined in the abstract, by which we mean that no specific implementations or technologies endorsed by it.

1. Overview 3
2. Microsystems 4
   1. System Descriptions 5
      1. Service Registry 5
      2. Authorisation System 5
      3. Orchestration System 7
   2. Local Cloud Bootstrapping 8
   3. The System-Service Matrix 9
3. Use-cases 10
   1. Service Discovery 10
   2. Authorisation 11
   3. Orchestration 12
   4. Authorisation and Orchestration 13
4. Control and Data Planes 14
5. Release Notes 15
6. Non-Functional Requirements 16
   1. Accurate Enough Message Caching 16
   2. Relative System Independence 16
   3. Backwards compatibility 17
   4. Authorisation mechanisms 17
   5. Naming convention for local clouds, microsystems, microservice, … 17
7. Unresolved Issues 17
8. References 18
9. Revision History 18
   1. Amendments 18
   2. Quality Assurance 18
10. Overview

The Eclipse Arrowhead project focus is to provide microservices which produces and consumes microservice at the edge[[1]](#footnote-2). Eclipse Arrowhead provides an microservice SOA architecture and reference implementation to be deployed on heterogeneous devices from deep edge to fog with interoperability to cloud servers.

The Eclipse Arrowhead project exists to produce solutions and specifications that support industrial and societal automation solutions that can adapt quickly and automatically to changing circumstances and business requirements. This adaptability is enabled through *Service-Oriented Architecture* (SOA), which entails dividing automation systems into smaller independent *systems* that communicate by sending messages to, or *consuming*, the *services* offered by the other systems. Each system is expected to be designed around the principles of

1. **loose coupling**, which means that services have narrow, well-defined application areas and depend internally on as few other services as possible to fulfil their functions,
2. **late binding**, which entails providing as much functionality as possible given what needed services, configuration data and other details are currently available, as well as
3. **lookup**, which means that systems, when possible, look up needed data by consuming services instead of relying on the data being in given configuration files.

These principles help facilitate the design of automation systems that are cheap and easy to implement, upgrade, extend and reuse, are highly resilient to failures and attacks, and are highly transparent about their constituents and activities.

While designing industrial automation systems around these principles has the advantages we mention, it also introduces the complexity inherent to all distributed applications. The Core microsystems of Eclipse Arrowhead exist to help manage this complexity by facilitating

1. **discovery**, or helping microsystems independently find the services they need,
2. **authorisation**, or regulating how microsystems are allowed to consume what microservices,
3. **authentication** of involved microsystem, as well as
4. **orchestration,** or centrally managing what microsystems consume what microservices.

Each of these concerns is addressed by its own microservice, provided by its own microsystem, as outlined in the following component diagram:

Each box in the diagram represents a microsystem. Every microsystem, except for the generic consumer, provides one mciroservice represented by an attached circle. Service consumption is denoted by forks extending from the consuming systems to the circles. Some forks are merged to improve clarity. A deployment of microsystems interrelated in this manner makes up a **S*ystem-of-Systems****, SoS*. When a System-of-Systems can fulfil its core functionality without external support, we refer to it as a ***local cloud***.

In the next section, Section 2, we consider what these services and systems do in more detail. In Section 3, we describe various use cases in which some or all of them are used. In Section 4, weDiagram

Description automatically generated discuss how the Core systems make up the *control plane* of a distributed application and why that matters. In Section 5, we outline important differences between versions 4 and 5 of Arrowhead. In Section 6, we present key non-functional requirements. In Section 7 we consider what parts of this document, which is still not officially released, are currently subject to discussion. Finally, in Sections 8 and 9 we list document references and versions.

1. Microsystems

The Core systems of the Eclipse Arrowhead project are meant to address concerns expected to be typical to most settings where SOA is applied. We describe them here in the abstract. The implementations of these systems are meant to be described in separate, non-generic documents referred to as *System-of-Systems Descriptions* (SoSDs).

|  |  |
| --- | --- |
| Microsystem | Microservice |
| Service Registry | Service Discovery |
|  | Service Registry Management |
| Authorisation System | Authorisation |
|  | Authorisation Management |
| Orchestration System | Orchestration Management |
|  | Orchestration Pull |
|  | Orchestration Push |

The Core systems, and the services they provide, are as follows:

The purposes of the respective Core systems are presented in the following subsection. In the subsection after that, we consider how local clouds end up in a working state when subject to a cold start. We then present a matrix of both core and support systems provided, or planned to be provided, by the Eclipse Arrowhead project.

Note that in addition to the Core systems, several *support* systems are described in the document *GSoSD Arrowhead Support Systems 5.0*. We expect also these support systems to be common in many Arrowhead deployments. Since they, however, are not as essential as the systems listed here, we keep them in a separate document to reduce the length and improve the clarity of this document.

# System Descriptions

We now proceed to informally state the purposes and services of the Core systems. More detailed abstract descriptions of the services they provide are available in separate documents we refer to as *Service Descriptions* (SDs).

## Service Registry

One of the main principles of SOA is *lookup*, which means that whenever possible, data required for whatever reason should be looked up by consuming a service. One kind of data that is virtually always required to facilitate a system-of-systems is the network information, such as IP addresses and port numbers, needed by systems to be able to communicate with each other.

The Service Registry holds a table that maps the identifiers of individual services to the information generally required to communicate with their providers. Individual systems may elect to register, update or deregister their own services in a Service Registry, and so make it possible for other systems to look them up. Alternatively, the Service Registry can be manually populated with entries representing the services that should be available.

When using a Service Registry, the only network address systems are required to know is that of the Service Registry. With that, the systems can look up the network addresses of the systems they need to consume—given that they know their type names or identifiers.

Entries in the Service Registry are not limited to holding only network address details. They may also include other metadata that may be relevant when selecting what specific service instances to consume. For example, it may be relevant to denote the physical locations of services that provide temperature and humidity measurements, or to indicate at what conveyer belts some welding service is provided.

Registering, updating, deregistering and querying the Service Registry is performed through its *Service Discovery* service. In addition, the system provides the *Service Registry Management* service, which allows for services to be registered, updated or deregistered in bulk. The latter service is meant to be useful primarily for administration.

## Authorisation System

When a system-of-systems applies any kind of access control, it becomes relevant for every service provider to know what systems are allowed to consume its services—and with what limitations. The Authorisation System holds a table that maps the identifier of each system to the identifiers of the individual services that system is permitted to consume, as well as additional details regarding any limits or tokens related to that consumption. Each entry in this table can be referred to as an *access control rule*. Whenever a system attempts to consume a service, the provider of that service may consult the Authorisation System to make sure that the consumer is permitted to do so.

Using an Authorisation System means that individual service providers do not need to know in advance what other systems are allowed to consume them.

To reduce the number of times a given system has to consult the Authorisation System, a *caching strategy* may be employed, as discussed further in Section [6.1](#Ref126571407).

Checking whether an attempted service consumption is allowed is performed through the *Authorisation* service of the Authorisation System. In addition, the system also provides the *Authorisation Management* service, which allows for access control rules to be added, updated, removed and queried. The latter service is meant to be useful primarily for administrative purposes.

## Orchestration System

Using only the Service Registry means that network information does not have to be manually supplied to the individual systems of a given local cloud. However, each system must still be manually configured, or use its own search procedure, to determine what exact service instances to consume. This becomes an issue when the same type of service is provided by many systems at the same time, but there are other differences to each provided service. If, for example, a given service instance reports the air quality for a specific location, what service instance is consumed affects what data is received by the consumer.

The Orchestration System holds a table that maps each relevant system to the specific service instances it *should* consume. Additionally, in contexts where tokens are used to enable access control, the Orchestration System also stores these tokens together with the identifiers of the services to be consumed.

The main task of the Orchestration System is to help the consumers to decide which service instances they should use. An orchestration request can result a list of providers that offers different services (to satisfy all the consumer’s needs); or a list of providers that offers the same service (viable alternatives); or just one provider.

There are multiple approaches how an Orchestration System can fulfil this task. All of them have different advantages and disadvantages.

The first approach (atomic service) holds a table that maps each relevant system to the specific service instances it *should* consume. The orchestration request returns service instance identifiers. The consumer can use these identifiers to acquire access information from the Service Registry. The advantages of this approach that is very simple, easy to implement, uses very few computing resources and the Orchestration System can work independently from other systems. The main disadvantage here is that this approach delegates most of the tasks to the consumer’s side. Also, the synchronization between the Service Registry’s data and the Orchestration mappings can be a challenge (for the local cloud administrator or a support system).

The second approach (advanced service) is more complex and offers a lot of features. It provides other types of orchestration besides the static orchestration rules: dynamic orchestration and a more flexible rule orchestration where rules are not peer-to-peer mappings but templates specified by system and service instance metadata. This solution requires the continuous assistance of the Service Registry and the Orchestration System cannot provide any useful information without it. If the local cloud contains other Core or Support systems, the Orchestration Core System can offer much more:

1. Authorization: if the consumer is not authorized the orchestration service removes the appropriate provider from the response;
2. Authorization: orchestration service automatically adds every necessary tokens (if the related provider requires it);
3. Gatekeeper: inter-cloud orchestration is possible;
4. Gateway (besides the Gatekeeper): inter-cloud orchestration is possible even between two closed local clouds and the necessary communication tunnel will be built during the orchestration process;
5. QoS Support: during orchestration Quality-of-Service requirements can be considered;

The advantage of this approach that is more feature rich, more convenient for the consumers (the orchestration response return everything what is needed for the consumer to perform a service consumption). The disadvantages here are that this approach is more difficult to implement, uses more computing resources and the system have dependencies to other systems (at least to the Service Registry).

There are use cases when the first approach are the preferable and there are use cases when the second one. So the Arrowhead framework should contain both orchestration systems with a shared interface and the operator of the local cloud should decide which kind of orchestration their local cloud should use.

Service consumers get lists of services to consume, or *orchestration rules*, by using the *Orchestration Pull* service of the Orchestration System. Consumers may also use the *Orchestration Push* service to be notified of any changes to their orchestration rules. In addition, the Orchestration System also provides the *Orchestration Management* service, which allows for orchestration rules to be added, updated, removed and queried. The latter service is meant to be useful for management and administrative purposes.

To address the late binding and loose coupled SOA properties "orchestrating" who can/should/shall talk to who is important. Thus the orchestration system provides rules on who can/should/shall talk to who. These rules may be based on policies.

Such rules can be "owned" by

* a consumer
* Plant description microsystem
* workflow/choreography microsystems
* orchestration microsystem
* operator - engineering
* …

The current hierarchy developed in Arrowhead is

1. Operator - Engineering
2. PlantDescription
3. Orchestration
4. Consumer

For interoperability reasons support for both push and pull distribution of orchestration rules should be supported. Each local cloud has it’s own orchestration system. In principle there can be multiple orchestration systems in a local cloud but that will create synchronisation requirement. Thus the recommendation is to have one Orchestration system in a local cloud.

The orchestration between local clouds is made possible using the Gatekeeper and Gateway microsystems in cooperation with ServiceRegistry and Authorisation/Authentication microsystems of other local clouds.

**It has to be noted that an application implementation where consumers are “free” to make use of their “own orchestration” information opens for less understanding and control of what’s going on in a local cloud, with possible consequences to e.g. real time, security and safety properties. Thus unforeseen issues my emerge.**

# Local Cloud Bootstrapping

When a local cloud starts up, none of its constituent systems will have the network addresses, credentials or other details necessary to consume the services it needs. For that data to become available for the systems to look up, the Core systems must first become ready to provide their services. We refer to the process of the Core systems becoming ready as *bootstrapping*. What is required for the bootstrapping of a particular local cloud to be successful depends on which of the Core systems are being used. However, one requirement is constant: *each system and every service must have a unique identifier prior to the bootstrapping process*.

If only the Service Registry is used, each system must be preconfigured with the network address of the Service Registry—or be able to determine it through some other well-defined mechanism. Every system must know how to decide what specific service instances to consume, something we consider more in Section [3.1](#Ref126321356).

If also the Authorisation System is used, all other systems must also be preregistered in its table, which maps system identifiers to the identifiers of the services each respective system is permitted to consume.

If also the Orchestration System is used, the need for having each system be able to independently determine what service instances to consume goes away. However, what systems are to consume what services must instead be registered in the table of the Orchestration System.

# The System-Service Matrix

En bild som visar bord

Automatiskt genererad beskrivningIn order to keep track of what services are produced and consumed by different systems, we present a system-service matrix below. This matrix outlines the Core systems in terms of what services they provide and consume.

The matrix is meant to eventually contain references to both the abstract definition of all systems, as well as to the available implementations of those systems. The abstract definitions are provided in documents we refer to as Service Descriptions (SDs), while the interfacing details of their implementations are documented in Interface Design Descriptions (IDDs). The full matrix can be found in Appendix 1.

1. Use-cases

To make it more apparent how the Arrowhead Core systems can be used in practical scenarios, we here present four use case descriptions. Each description centres around a sequence diagram illustrating how systems consume each other’s services.

# Service Discovery

Diagram

Description automatically generatedThe first, and most basic, of our use cases shows how some system **B** can consume the service **a**, provided by **A**, by looking up its address in a Service Registry instance. No security mechanisms are in place. The only precondition is that **A** and **B** both know of the network address of the Service Registry instance. The use case is depicted below.

The use case consists of three message exchanges, which are as follows:

1. **A** registers its service **a** with the Service Registry, which accepts the registration.
2. Later, **B** attempts to lookup the service **a** via the Service registry, which responds with the network address, and other details, of **a**.
3. **B** then proceeds to consume **a** by sending a request to **A**, which responds with some status and payload of relevance to the service.

As we assumed that only one instance of the **a** service was registered in the Service Registry, the response in exchange 2 only contained one address. If, however, more than once such service instance would have been registered, **B** would have received the network addresses of all of them. **B** would then have been forced to determine by itself which instance would be most appropriate to consume, perhaps by looking at other details also present in the received registrations.

Alternatively, **B** could have provided the instance identifier of a particular service instead of a service type name in its query request to the Service Registry. Since those identifiers must be unique, the query result could then only contain at most one result. It would, however, also mean that relevant service instance identifiers must be known in advance, and that the specific instance is online when **B** needs to consume it.

Another approach is to centralise decisions about service consumption using an Orchestration System, as shown later in Section [3.3](#Ref126315635).

# Authorisation

Diagram

Description automatically generatedIn this, our second, use case, we add access control by introducing an Authorisation System. Whenever an attempt is made to consume a service, its providing system validates the request by consulting the Authorisation System before responding to it. We assume that all services were registered in the Service Registry before the use case begins. We also assume that the Authorisation System has already been provided with an appropriate set of access control rules. The use case is depicted below:

The use case consists of the following message exchanges:

1. **A** looks up **b** via the Service Registry, which responds with the network address of **b**.
   1. However, to make sure A is authorised to discover services, the Service Registry consults the Authorisation System before responding, which gives its approval.
2. **A** attempts to consume **b** by sending a request to **B**, which later responds.
   1. Before responding, however, **B** must ascertain that **A** is permitted to consume its **b**. Since **B** does not yet know the network address of the Authorisation System, it queries it and subsequently receives the address.
      1. In turn, the Service Registry checks if B is permitted to consume its Service Discovery service, which it proves to be.
   2. **B** then proceeds to check with the Authorisation System if **A** is permitted to consume its instance of **b**, which reports that it is.

Using authorisation in this manner means that a list of access control rules must be prepared and provided to an Authorisation System in advance. As such rules must contain the unique identifiers associated with the various systems and services in the given local cloud, also those identifiers must be known in advance. To be able to guarantee that every system is associated with the expected set of identifiers, it must be preconfigured with its own set of identifiers. It is possible to avoid this need for preconfiguration by using a separate system that dynamically creates access control rules according to given strategies. Such a system is, however, beyond the scope of this document.

# Orchestration

As we mentioned towards the end of the first use case in Section [3.1](#Ref126321356), using only a Service Registry means that we must either rely on (1) systems being able to choose what specific service instances to consume, or (2) being able to preconfigure each system with the identifiers of those specific service instances. If we, on the other hand, introduce an Orchestration system, we can avoid either by having the individual systems lookup what services we *want* them to consume.

Chart, box and whisker chart

Description automatically generatedWe assume that all services were registered in the Service Registry before the use case begins, as well as that the Orchestration System is using the advanced approach and has been provided with an appropriate set of orchestration rules, which describe what systems should consume what services. We also assume no security mechanisms are in place. The use case is depicted below:

The use case consists of the following message exchanges:

1. **A** looks up the network address of the Orchestration Pull service by consuming the Service Discovery service of the Service Registry, which responds with the requested address.
2. **A** queries the Orchestration System for the network addresses of the services it should consume, which responds with the network address of **b**, provided by **B**.
   1. Since the Orchestration System does not hold any actual network addresses, it consults the Service Registry for the network address of **b**, which it also gets.
3. **A** consumes **b** of **B** by sending some request, which **B** responds to.

In this particular use case, orchestration information was *pulled* by **A**, by which we mean that A requested the information from the Orchestration System on its own accord. The Orchestration System also supports *pushing* orchestration, however, which means that rather than requesting it directly, **A** could have subscribed to changes to the orchestration relevant to itself. Pushing orchestration rules in this manner means that changes to the rules are more likely to end up at their respective systems. They are also likely going to get the information quicker than if they had been requesting orchestration from the Orchestration System at regular intervals.

# Authorisation and Orchestration

While we already covered authorisation and orchestration separately in Sections [3.2](#Ref126322842) and [3.3](#Ref126315635), we have one important reason to consider them being used at the same time. The Orchestration system can create (using the Authorization System) and distribute access tokens, which can be used to increase security by having the credentials actually passed between systems expire faster.

Diagram

Description automatically generatedWe assume that all services were registered in the Service Registry before the use case begins. We also assume that **A** has already looked up the network address of the Orchestration System, as well as that the Orchestration System is using the advanced approach and has been provided with an appropriate set of orchestration rules. We also assume that the Authorisation System has already been provided with an appropriate set of access control rules. The use case is depicted below:

The use case consists of the following message exchanges:

1. **A** requests its set of orchestration from the Orchestration System, which later responds with the network address of **b** and the access token **A** needs to consume **b**.
   1. First, however, the Orchestration System checks if **A** is authorised to request its orchestration, which it proves to be.
   2. Then the Orchestration System determines the network address of **b** by consulting the Service Discovery service of the Service Registry.
      1. The Service Registry checks if the Orchestration System is allowed to consume the Service Discovery service, which it is.
   3. Before responding A, the Orchestration System also requests the Authorisation System to create an access token for A, which it does.
2. **A** consumes **b** and includes the access token it received in the request. **B** responds.
   1. Before responding, **B** uses the access token to check if **A** is authorised to consume **b**, which it proves to be.
3. Control and Data Planes

None of Core and Support systems provided by the Eclipse Arrowhead project are designed to directly fulfil the business objectives of any given local cloud. What they are designed to do, however, is to help create the situation in which these business objectives can be fulfilled. They do this, generally, by controlling and distributing information *about* the local cloud and its resources.

Traditionally, computer networks are reasoned about in *planes*, where each plane presents the same network in terms of a different perspective with its own aims and conditions. The term *control plane* is used when considering how networking equipment, such as firewalls, switches and routers, are setup to allow for information to flow in certain ways. The term *data plane*, on the other hand, is used when considering the contents of those information flows—without regard to how they were set up. This separation into planes increases the number of things that must be considered, but it also reduces the complexity of each of those things.

In a similar manner, we use the *term control* plane when considering how systems are controlled to facilitate desired service consumption patterns. In a control plane component or sequence diagram, for example, interactions between Core, Support and application systems are illustrated, but communications between application systems are left out. The reverse is the case for *data plane* diagrams: interactions with and between Core and Support systems are generally omitted, while interactions between application systems are included.

In this document, we do not make a distinction between control and data plane diagrams. While we certainly do focus on our control plane, the data plane is included for the sake of clarity. To get a better understanding of what control and data plane diagrams could like, consider the sequence diagram in Section [3.4](#Ref126576068). Its message exchange denoted with ❶ is exclusively concerned with the control plane. In it, the application system **A** gets the control plane details needed for it to consume the service **b**. Also, the *a* exchange of message sequence ❷ belongs to the control plane, as it is between a Core system and an application system. A pure data plane version of the diagram would only include the message exchange between systems **A** and **B**, without any references to access tokens or other control plane details.

An important aspect of the control plane is that no data exchange is allowed to occur here. The core systems should not have to consider the requirements of the data transmissions in the Arrowhead cloud. Furthermore, once the system-of-systems have been established and no need for change is imminent, the control plane could even be removed or shut down. Keeping a separation of the control and data plane allow different performance and different scaling of respective plane.

When designing your local cloud, it is generally useful to split your design into a control and a data plane. The first of the two would be concerned with which Core and Support systems, as well as other complementary systems, are being used and under what conditions. The second of the two would be concerned with how your local cloud directly addresses its business objectives.

1. Release Notes

**Work in progress.**

This section describes the differences between current version and previous versions of the architecture. It will be written in full when the issues in Section 7 have been resolved.

1. Non-Functional Requirements

Setting up and operating an Arrowhead local cloud comes with the many caveats shared by all distributed systems. Managing these caveats typically means that you need to prioritise certain qualities when designing or choosing the systems for your local cloud. In this section, we outline such qualities promoted by the Eclipse Arrowhead project as a list of non-functional requirements, one in each of the following subsections.

# Accurate Enough Message Caching

Most modern computing devices, including embedded such, can call local functions in the order of microseconds or faster. In contrast, calling a function on another device by sending a message is typically completed in the order of milliseconds or slower. The relative expense, in time and capacity utilisation, of sending messages is often high enough to often make it worthwhile to spend quite some effort to reduce the number of messages that must be sent for the use case at hand to be possible to realise.

One important way to reduce messaging is to make it possible for systems to *cache* data they receive, by which we mean that the data is saved with the expectation that it may become useful in the future. When a system finds itself in need of some data of concern, it first consults its cache to see if it can find it there. If it can, it uses that data, and no message needs to be sent.

The trade-off inherent in using caches is, however, that the entries they contain become invalid as soon as the original data they are based on change. Unless the caching systems are notified about the event, they cannot be confident the original data has not changed unless they request it again, which requires sending the message we wanted to avoid in the first place. If notifying systems is not practical for a given use case, then there are many ways to have cache entries expire when too old or when other factors make it seem as if they may have become invalid. These ways are beyond the scope of this document, however.

The important thing to note here is that caching can indeed help reduce the need for sending messages, but using it requires carefully planning how to avoid systems acting on invalid cache entries. The Arrowhead Core and support systems must all be designed to make it easy to reason about and manage the caching of the data they hold.

# Relative System Independence

Systems tend to become more brittle the larger and more complicated they become, and local clouds are no exception. The more and the more complicated systems make up a local cloud, the larger the probability gets of any of those systems degrading or breaking. One way to mitigate this is by promoting systems with small scopes, which means fewer things can break in each system, as well as by making the systems able to independently work around these kinds of eventualities. However, making systems independent means that each of them needs to own enough resources and capabilities for this independence to be possible. Among other things, this entails systems owning their own data rather than storing it in external databases.

Designing a local cloud effectively means that its resilience to failures must be accounted for, and that its systems are given the capabilities required to work around any contingencies.

# Backwards compatibility

For major realises some backwards incompatibility may occur. From v4.x.y to 5.00 the implementation of the core and support systems shall be responsible for their own data. The v4.x.y implementation used a common SQL data base for common data which will not be the case for 5.0.0 and onwards. Thus v5.x.y. core systems will not be backwards compatible with the current release v4.6.1 and its predecessors.

The general approach to backwards compatibility follow below guidelines:

The current strategy is that we are moving to the next major version 5.0  
A v5.x+1.y will be release based on a new core system being added or a major update of one of the release core systems. A v5.x.y+1 release is bug fixes.

The current backwards compatibility and version maintenance approach is:  
v4.6.1 will be maintained for at least 12 months after the release of v5.0.0  
v5.1 should be backward compatible with v5.0  
v5.2 should be backward compatible with v5.0 and v5.1  
v5.3 should be backward compatible with v5.0, v5.1 and v5.2

# Authorisation mechanisms

Eclipse Arrowhead has te ambition to support a number of authorisation mechanisms. For v5.0 the major approach will be X.509 certificates. For upcoming versions the addition of other authorisation mechanism like OAuth2.0, OpenID, etc. will be investigated.

# Naming convention for local clouds, microsystems, microservice, …

It has been agreed that a naming convention should be developed. The proposal buy Cristina Paniagua and Jerker Delsing from 2019 is considered to complex. Thus a discussion in eclipse-arrowhead/roadmap/issues/69 is initiated to for a robust and simplified naming convention for Eclipse Arrowhead local clouds, microsystem, microservice, ….

1. Unresolved Issues

As far as is currently known, the following issues remain to be addressed before this document can be released as version 1.0.

All point of major discussion #1 - 10, see issues in the roadmap directory of Github have reached acceptable conclusions.

1. References

**Work in progress.**

1. Revision History

# Amendments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Date | Version | Subject of Amendments | Author |
| 1 | 2023-01-12 | 0.1 | Initial draft. | Per Olofsson |
| 2 | 2023-02-03 | 0.2 | Refined and extended Sections 1, 2 and 3. Removed unneeded sections. Cleaned up the other. | Emanuel Palm |
| 3 | 2023-02-07 | 0.3 | Update figures. Update Sections 1, 2. Write Sections 4, 6 and 7. | Emanuel Palm |
| 4 | 2023-02-24 | 0.4 | Added to Section 4. | Per Olofsson |
| 5 | 2023-09-24 | 0.5 | Updated according to Issue #3,5,7,8 | Jerker Delsing |
| 6 | 2023-09-25 | 0.6 | Updates according to Issue #2 | Raymond Bocsi |

# Quality Assurance

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Date | Version | Approved by |
| 1 | YYYY-MM-DD | 1.0 | Nnnnn Nnnnnnn |
| 2 |  |  |  |

1. The terms microsystem and microservice will be used interchangeable with system and service to simplify writing. [↑](#footnote-ref-2)