## Deliverable D1.5

# The SmartCLIDE Architecture

## WP 1

|  |  |
|---|---|
| **Project Acronym & Number:** | SmartCLIDE – GA 871177 |
| **Project Title:** | Smart Cloud Integrated Development Environment supporting the full-stack implementation, composition and deployment of data-centered services and applications in the cloud |
| **Status:** | Final |
| **Dissemination Level:** | Public |
| **Authors:** | INTRA |
| **Contributors:** | ALL PARTNERS |
| **Document Identifier:** | SmartCLIDE-D1.5 Architecture v1 |
| **Date:** | 09.11.2020 |
| **Revision:** | 1.0 |
| **Project website address:** | https://smartclide.eu |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No *871177*

## Partner Contacts

**Institut für angewandte Systemtechnik Bremen GmbH (ATB), Germany**
Intrasoft International SA (INTRA), Luxembourg
Fundacion Instituto Internacionale de Investigacion en Intelligencia Artificial y Ciencias de la Computacion (AIR), Spain
University of Macedonia (UoM), Greece
Ethniko Kentro Erevnas Kai Technologikis Anaptyxis (CERTH), Greece
X/OPEN Company Limited (TOG), United Kingdom
Eclipse Foundation Europe GMBH (ECLIPSE), Germany
Wellness Telecom SL (WT), Spain
Unparallel Innovation LDA (UNP), Portugal
CONTACT Software GmbH (CONTACT), Germany
Kairos Digital, Analytics and Big
Data Solutions SL (KAIROS DS), Spain

## Dissemination Level

| | | |
|---|---|---|
| PU | Public | X |
| PP | Restricted to other programme participants (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium (including the Commission Services) | |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |

## Document Control

| Version | Notes | Date |
|---|---|---|
| 0.1 | Creation of the document | 25/08/2020 |
| 0.2 | Initial contributions and architectural diagrams by INTRASOFT, partner contribution assignments | 18/09/2020 |
| 0.3 | Update of the ToC and partner assignments, initial partners contributions | 07/10/2020 |
| 0.8 | Update of architectural diagrams and descriptions, further partner contributions | 30/10/2020 |
| 0.9 | Alignment of descriptions, finalization for internal review | 31/10/2020 |
| 1.0 | Internal review, finalization for formal submission | 09/11/2020 |

## Abbreviations

| | |
|---|---|
| AB | Advisory Board |
| AI | Artificial Intelligence |
| API | Application Programming Interfaces |
| App | Software Application |
| APM | Adaptive Project Management |
| BFF | Backend for Frontends |
| BPMN | Business Process Model and Notation |
| D | Deliverable |
| DLE | Deep Learning Engine |
| DoA | Description of Action |
| EA | Ethical Adviser |
| PB | Plenary Board |
| EC | European Commission |
| e.g. | exempli gratia = for example |
| etc. | et cetera |
| EU | European Union |
| FP7 | Framework Programme 7 |
| GA | Grant Agreement |
| GDPR | General Data Protection Regulation |
| ICT | Information and Communication Technology |
| i.e. | id est = that is to say |
| IP | Intellectual Property |
| IPR | Intellectual Property Rights |
| LAN | Local Area Network |
| JSON | JavaScript Object Notation |
| KPI | Key Performance Indicator |
| M | Month |
| PB | Plenary Board |
| PC | Project Coordinator |
| PQA | Project Quality Assurance |
| QA | Quality Assurance |
| QoS | Quality of Service |
| REST | REpresentational State Transfer |
| RPI | Remote Procedure Invocation |
| RTD | Research and Technological Development |
| SME | Small and Medium Sized Enterprise |
| SC | Steering Committee |
| SOAP | Simple Object Access Protocol |
| STQA | Scientific and Technical Quality Assurance |
| T | Task |
| UI | User Interface |
| UML | Unified Modeling Language |
| VoIP | Voice over IP |
| WP | Work Package |
| WPL | Work Package Leader |
| WPMT | Work Package Management Team |
| w.r.t. | with respect to |

# Executive Summary

This document presents the architecture of the SmartCLIDE system. It is the result of the design process, strongly dependent on and complementing the defined SmartCLIDE requirements, use cases and conceptual design of its components. Consequently, taking into account both the results of requirements, the specified set of system use cases and pilot scenarios that captured in detail how the envisioned SmartCLIDE system will offer its functionality to the users, and the envisioned technical innovations of the SmartCLIDE system as outlined in its conceptual design, this document focuses on detailing the component-based architecture, the information flows and component interactions view, as well as the deployment architecture of SmartCLIDE. The architecture description is also complemented by the delivery plan of the system - the approach to be used and a time plan for the delivery of the system with specific phases and milestones has been included in the delivery plan.

All the aforementioned content has been structured following the concept and terms of the ISO/IEC/IEEE 42010:2011, “Systems and software engineering — Architecture description” standard. Although this document does not fully comply with the standard’s requirements, the use of the principles included in the standard increases the standardization of the architecture description and the readability of the document itself. It must be noted that the design process and architecture specification of the system is an ongoing process that will continue in the next phases of the project on the light of new deliverables that are due in the months to follow. Thus, even though the current document, along with D1.3 and D1.4, is a starting point for the specification of the system, modifications will apply in the course of the project in order to record the evolving requirements and the corresponding changes in the architecture specification, following an agile development approach.

# Table of Contents

## List of Figures

## List of Tables

# 1 Introduction

## 1.1 Document Purpose

Deliverable D1.5 “The SmartCLIDE Architecture” is part of WP1 and is produced as the main outcome of Task 1.5: Design of SmartCLIDE Architecture. The purpose of this document is to provide an architectural design of the SmartCLIDE platform, from different architectural viewpoints, that have not yet been covered by other WP1 Deliverables, based on the outcomes of the other WP1 Deliverables, and more specifically D1.2: “Requirements Analysis”, D1.3: “Use Case Scenarios” and D1.4: “The SmartCLIDE Concept”.

## 1.2 Approach

A micro services-oriented architecture is envisioned following a Service Oriented Modelling approach. It will allow deployment of services in containers within the cloud, composed of horizontal services (Deep Learning Engine, Back-end features like micro-services storage and discovery with knowledge search engine, data persistence, or user management, as well as system security) as well as vertical services (Development, Testing, Deployment and Runtime simulation).

Service-oriented modelling (SOM) is a discipline on top of which several popular approaches for modeling business and software systems have been arisen. Its purpose is to design service-oriented business systems without limited to a specific domain or architectural style. Examples include microservices, cloud computing and the more traditional application architecture, service-oriented architecture, and more.

SOMF is used to create models though-out all the Software Development “phases”, from analysis to design and architecture and embraces the simplicity of the representation that it provides, which is ideal for individuals with diverse level of business and technical background. All Service-oriented modeling approaches typically include a modelling language that can come from both the “problem domain organization” and the “solution domain organization”. It acts as a “reference” for the involved parties to enhance the strategy that is to be followed from the analysis phase.

The SmartCLIDE architecture specification will combine several established architectural and modeling approaches by leveraging the foundations, principles and disciplines of service-oriented modeling. This will result in a hybrid model that will describe the reference architecture (conceptual or logical where applicable) for all defined Architectural Views. Architecture diagrams will also be presented including Component / Composite Structure, Information / Data Flows and Deployment diagrams, taking input from the component design specifications in D1.4: “The SmartCLIDE concept”.

### 1.2.1 The Microservices architectural approach

As an alternative to the somewhat traditional Monolithic architectural approach for developing applications, which under certain conditions can still remain a good choice, the Microservices architectural approach represents an option which not only addresses limitations and issues of the former, but also is a good fit for large and complex applications.

The Microservices approach achieves this by adopting a strategy of putting together a large and complex application from small individual building blocks. These are distinct components that perform a specific function, examples of which include, among others, processing of data, login services and persistence of information. These individual and discrete components (microservices) can be considered as separate software components which have their own code and resources. The entire functionality of the system is therefore realized and composed by the microservices available, which work together as a whole, communicating among them and clients of the system to fulfil requests.

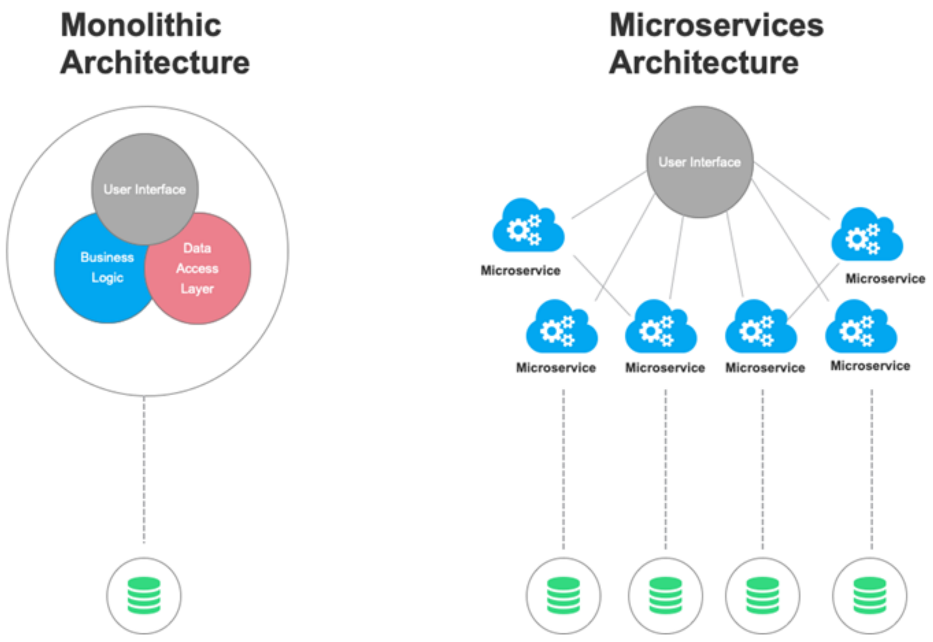


**Figure 1: Monolithic vs Microservices architecture conceptualization (found also in D1.1)**

The overall aim is to structure an application as a collection of services that are: highly maintainable and testable, loosely coupled, independently deployable, organized around business capabilities and each owned by a small team. It is expected that each small service is running in its own process and communicates with other services and clients using lightweight mechanisms such as well-defined HTTP resource APIs (web services). These APIs frequently and depending on the case employ authenticated and encrypted communications that follow the RESTful paradigm (further discussed in later sections of this document). The services are implemented in ways that allows for them to be deployed independently from each other while always having in mind fully automated deployment approaches such as CI/CD. This particular approach further allows for minimum centralized management of the available services that can also be implemented by different teams, using different programming languages and potentially employ different data storage approaches if necessary.

## 1.3 Document Structure

The deliverable consists of the following sections:

- Section 2 describes the different views of a System Architecture, detailing which views have already been presented in other WP1 Deliverables, namely, D1.2: “Requirements Analysis”, D1.3: “Use Case Scenarios” and D1.4: “The SmartCLIDE Concept”, and which are to be presented in the current Deliverable
- Section 3 presents the Conceptual / Component-based Architecture diagram in detail
- Section 4 presents the Information Flows / Communication Architecture diagram, along with an initial definition of the envisioned component interfaces and exchanged messages with other interfacing components
- Section 5 presents the System Deployment Diagram, providing also details as to the current deployment contexts of the envisioned SmartCLIDE pilots
- Section 6 presents the SmartCLIDE System Delivery Plan along with the methodology to implement it.
- Section 7 outlines the conclusions.

## 1.4 Relationship to other Deliverables

The present Deliverable completes the design and architecture specification of the SmartCLIDE platform from all design and architecture viewpoints, together with D1.2, D1.3 and D1.4. Essentially it receives inputs from all these Deliverables and primarily from D1.4, while it provides inputs, along with D1.2, D1.3 and D1.4, to the subsequent implementation tasks in WP2 and WP3.

All the aforementioned Deliverables content has been structured following the concept and terms of the ISO/IEC/IEEE 42010:2011, “Systems and software engineering — Architecture description” standard. Although this document does not fully comply with the standard’s requirements, the use of the principles included in the standard increases the standardization of the architecture description and the readability of the document itself.

## 1.5 Contributors

All technical partners contributed significantly to the design process. We also received contributions from use case partners in terms of current deployment best practices.

# 2 Architecture Views

Views define the various perspectives from which the system must be described in order to provide a complete architecture description to the stakeholders of the project. As such each view of the system is the actual representation of the system from that perspective. In other words, the view can be explained as a “where the system is been looked from”/”how the system looks from there” pair. The architectural views are multiple, with the most significant being the following, some of which have already been addressed in Deliverables D1.2, D1.3, and D1.4, while the rest are presented in the current Deliverable:

- **Functional requirements view:** this view should describe the functional specification of the various components of the system. A list of non-functional requirements could also be presented in such a view. The Functional requirements views have already been collected, analyzed and detailed in D1.2: “Requirements Analysis” and mapped also to SmartCLIDE System Use cases in D1.3: “Use Case Scenarios”.
- **Components specification view:** this view presents the detailed description of each component of the system. Each description includes the components role in the system, its internal design, quality attributes and APIs or User interfaces if applicable. This view has already been presented in D1.4: “The SmartCLIDE Concept”.
- **Operational environment view:** this view describes the conceptual way the system operates in its designated context, the components that comprise the system, where these components are deployed and the user groups with which they interact. These have already been detailed in D1.3 and D1.4. A more detailed depiction of the conceptual/logical architecture, in terms of a Component Architecture Diagram of SmartCLIDE is also further included in this Deliverable.
- **Information flow view:** this view defines the information entities handled by the system as well as the corresponding relationship between them. It also includes the flows (exchange) of the information between the system users and its components as well as between the components themselves. This view will be described in the present Deliverable.
- **System deployment view:** this view will present the deployment view of the integrated system and, per each pilot, the current approach and available infrastructure.
- **Delivery view:** this view will describe a delivery time plan for the delivery of the components and for the platform as a whole, along with a description of the approach for its implementation.

Concluding, given that architecture and design specifications from different views have in part already been presented in other WP1 SmartCLIDE Deliverables (D1.2, D1.3, D1.4), this Deliverable (D1.5) will present in detail only the conceptual/logical architectural view, the information flows/communication view, the deployment view and the delivery plan.

# 3 SmartCLIDE Conceptual System Architecture – Operational view

This section presents the interactions among SmartCLIDE modules in the form of a UML component diagram. The majority of SmartCLIDE activities take place at the Service Composition and Testing (SCTM) module where users search, select and compose services from different sources. In other words, this module hosts the “service laboratory” where new business flows are developed in the special business flow canvas with the help of the embedded Process Engine. Therefore, it is also the only responsible module for the Visualization, Simulation, Testing and Deployment of services because the application format can only be executed by the Process Engine which exists only in SCTM. For the discovery of existing services, the SCTM depends on the Service Discovery (SDM) component, as shown in Figure 2.

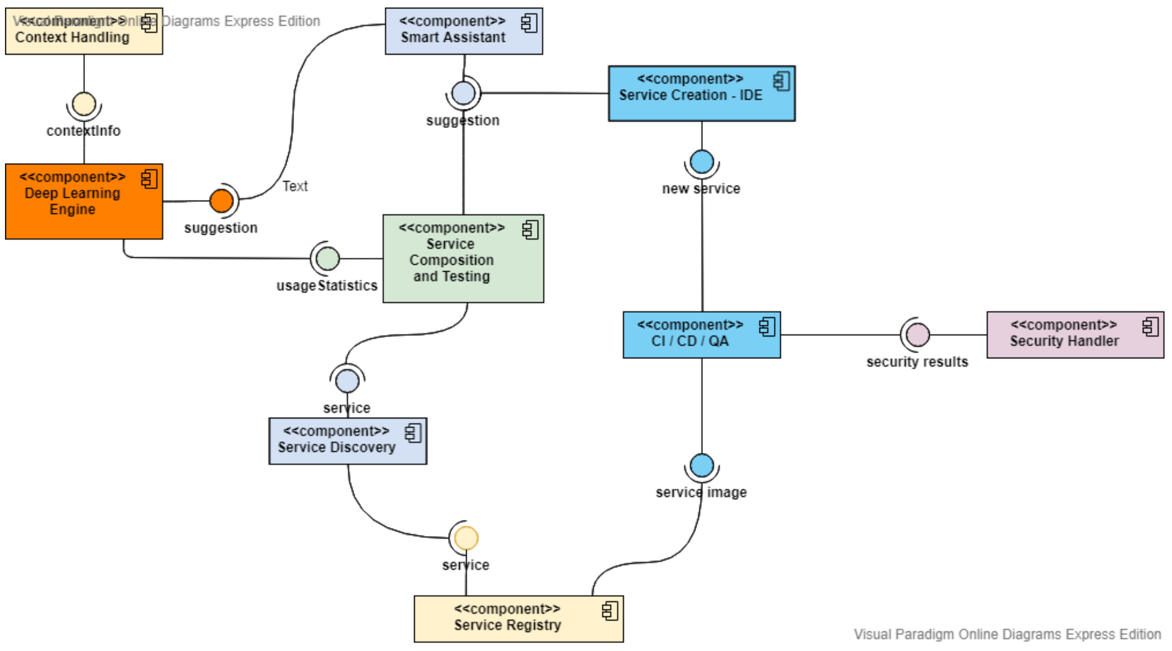

<<component>> Context Handling
contextInfo
<<component>> Deep Learning Engine
suggestion
Text
<<component>> Smart Assistant
suggestion
<<component>> Service Creation - IDE
new service
<<component>> Service Composition and Testing
usage Statistics
service
<<component>> Service Discovery
service
<<component>> Service Registry
<<component>> CI / CD / QA
security results
<<component>> Security Handler
service image
Visual Paradigm Online Diagrams Express Edition

**Figure 2: Component-based SmartCLIDE Architecture**

The creation of new services is the responsibility of the Service Creation - IDE (IDE) module. The services need to be containerized first. Therefore, the source code passes through the CI/CD module where also Quality Assurance, as well as filtering and security analysis (by the Security Handler module - SECM) are applied. If all necessary QA and Security requirements are met, the service image is created and pushed into the Service Registry (SREGM) which acts as the feed for the Service Discovery.

Activity regarding all used services is been constantly fed to the Deep Learning Engine Module (DLEM) which is being trained facilitated by the Context Discovery (CDM). Both the creation and composition of services get recommendations provided by the Smart Assistant Module (SAM) which materializes the knowledge created from the Deep Learning Engine.

Regarding the inter-module communication means, to ensure a decoupled, scalable and fault-tolerant approach, a Message Oriented Middleware will be used for one-to-many or many-to-many communication needs.

# 4 SmartCLIDE Information Flow View

## 4.1 Information Flows – Communication Diagram

This section presents the information flow view for the core components of the SmartCLIDE platform in a simplified form (Figure 3) for readability purposes, as a UML Information Flow diagram. This view focuses both on the component-to-component interactions and on user-to-system ones. The exchange method of the information between components is not depicted and it may vary between various components.

Following Requirements Analysis presented in D1.2, Use Case Scenarios definition presented in D1.3 and Concept presented in D1.4, SmartCLIDE can be described as a platform composed by 3 major high-level components:

1. SmartCLIDE User Interface
2. SmartCLIDE Back-end Services
3. 3rd party Solutions/Infrastructure interacting with SmartCLIDE through API Interfaces

The first two can be further decomposed in the sub-components which are the core SmartCLIDE platform components and are described below. For some of these core components more details can be found about their internal information flow and how this impacts the overall flow of the system.

As SmartCLIDE platform consists of multiple components that need to interact with each other and require different information exchange methods, protocols for communication, data formats, etc. certain architectural patterns have to be applied for the ease of design, implementation, maintenance and further improvements.

Two main scenarios need to be satisfied, that of course can be further decomposed. The first scenario requires Back-end Services or any other internal component to expose their APIs to User Interface components, or any other actor outside the internal system. The second scenario requires Back-end Services or any other internal component to interact with each other. As these two general scenarios are based on different requirements, demand a different design approach.

For the first scenario, API Gateway pattern is applied. A component acts as the single-entry point for the User Interface components or external actors. The API gateway handles requests in one of two ways. Some requests are simply proxied / routed to the appropriate service or others are handled by fanning out to multiple services. In this component, all functionalities related to an API Gateway take place. Routing, data transformation, load balancing (where applicable), protocol transformation, security (authentication/authorization/token flows), etc. are provided. The typical API Gateway pattern is applied, without limiting for future updates to adapt a more flexible variation of the pattern like BFF (Backends for frontends) by defining separate API Gateways, if there is a requirement to further decompose communication flows. By applying this pattern, repetitive actions are performed in a central place and with respect to Consumer Driven Contracts principle, apply all the data transformation required in a common manner limiting the need for adapters on the component level. As such, SmartCLIDE IDE can get access to all data required from the underline system.

For the second scenario, internal component communications are treated either synchronously or asynchronously.

In the case of asynchronous communication, a Messaging pattern is applied. A component acts as the single-entry point for all internal asynchronous communications that take place among the Back-end services. This enhances the collaboration between internal services, components, etc, limiting when required the burden of synchronous communication that results in tight runtime coupling. The communication can take place on messaging channels (one or more, can be defined appropriately) and various architectural styles of asynchronous communications can be applied per case, based on the component requirements. The main architectural style that is applied is Publish / Subscribe without of course limiting for future updates, to further enhance the system if required by introducing also Publish / Asynchronous response, Notifications, Request / Asynchronous response, Request / Response, etc. By applying this pattern, various communication protocols can be supported that offer the ability to establish and maintain network conversation between components to exchange real-time data. As such Context Handling Component for example can subscribe to Runtime Monitoring and Verification data for getting real-time monitoring information, Deep Learning can retrieve real-time context information, etc.

In the case of synchronous communication, Remote Procedure Invocation (RPI) pattern is applied. Internal components can still communicate directly with-each other in a more traditional manner through the APIs exposed from the components themselves offering a simple, familiar, request/reply, with no intermediate communication. The actual RPI technology applied (REST, gRPC, etc.) in this case, is up to each component to provide.

The above patterns enhance the Separation of Concerns principle, scalability and performance, request orchestration, response translation, fallback enablement, rate limiting and access control, autonomy, loose coupling, etc.

A more detailed and elaborate  Information Flows diagram is presented in Figure 4 but due to the level of detail and the complexity of the diagram, it is not easily readable within a pdf Deliverable. However, it is included in the present Deliverable to demonstrate the work performed and base the descriptions of the information exchanges among components that succeed in this subsection.

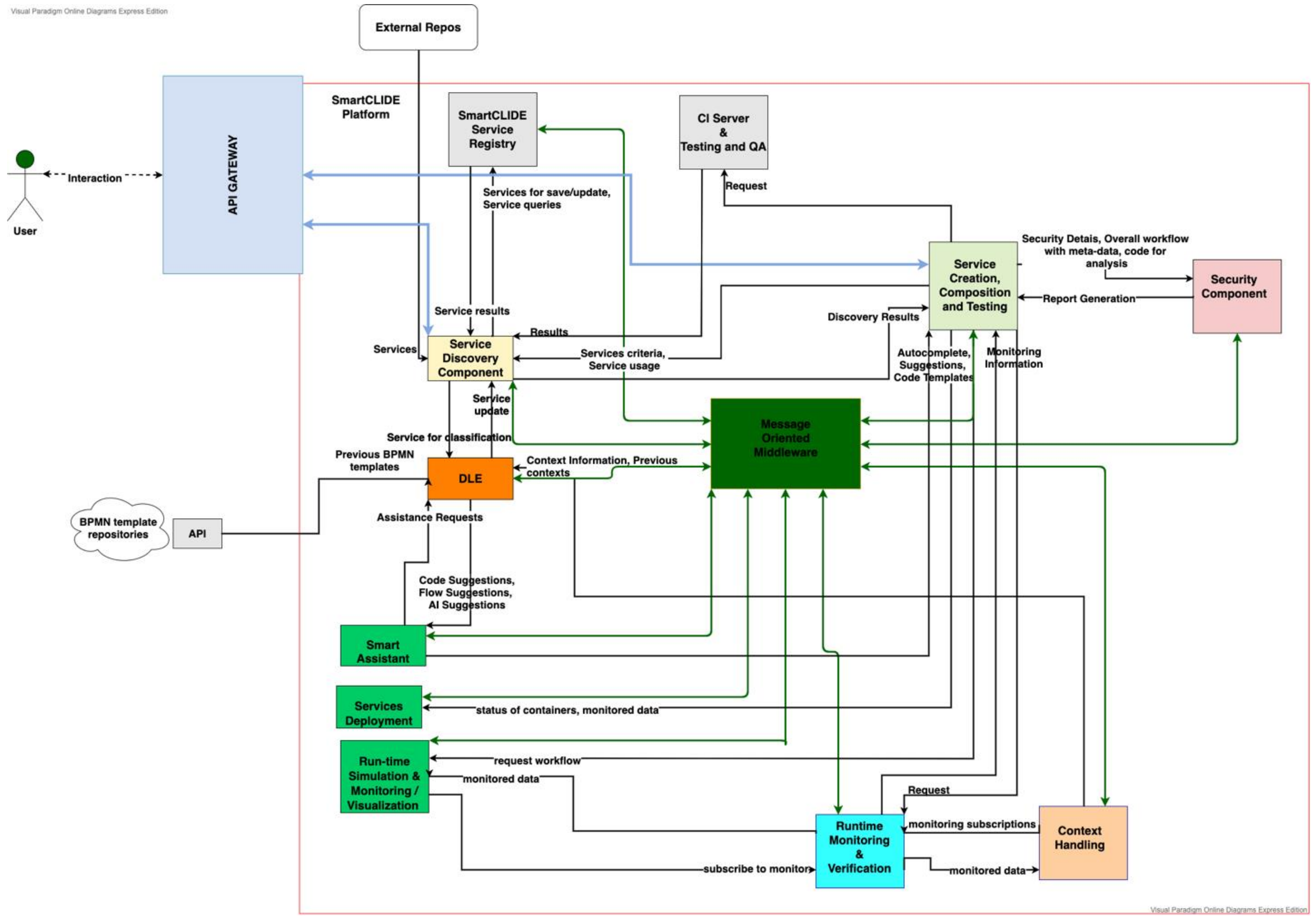

Visual Paradigm Online Diagrams Express Edition
External Repos
SmartCLIDE Platform
SmartCLIDE Service Registry
CI Server & Testing and QA
API GATEWAY
User
Interaction
Services for save/update, Service queries
Request
Service Creation, Composition and Testing
Security Details, Overall workflow with meta-data, code for analysis
Security Component
Report Generation
Service results
Results
Services
Service Discovery Component
Services criteria, Service usage
Discovery Results
Autocomplete, Suggestions, Code Templates
Monitoring Information
Service update
Message Oriented Middleware
Service for classification
Previous BPMN templates
DLE
Context Information, Previous contexts
BPMN template repositories
API
Assistance Requests
Code Suggestions, Flow Suggestions, AI Suggestions
Smart Assistant
Services Deployment
status of containers, monitored data
Run-time Simulation & Monitoring / Visualization
request workflow
monitored data
Request
Runtime Monitoring & Verification
monitoring subscriptions
Context Handling
subscribe to monitor
monitored data
Visual Paradigm Online Diagrams Express Edition

**Figure 3: SmartCLIDE Information Flows – Communication Architecture – Simplified Version**

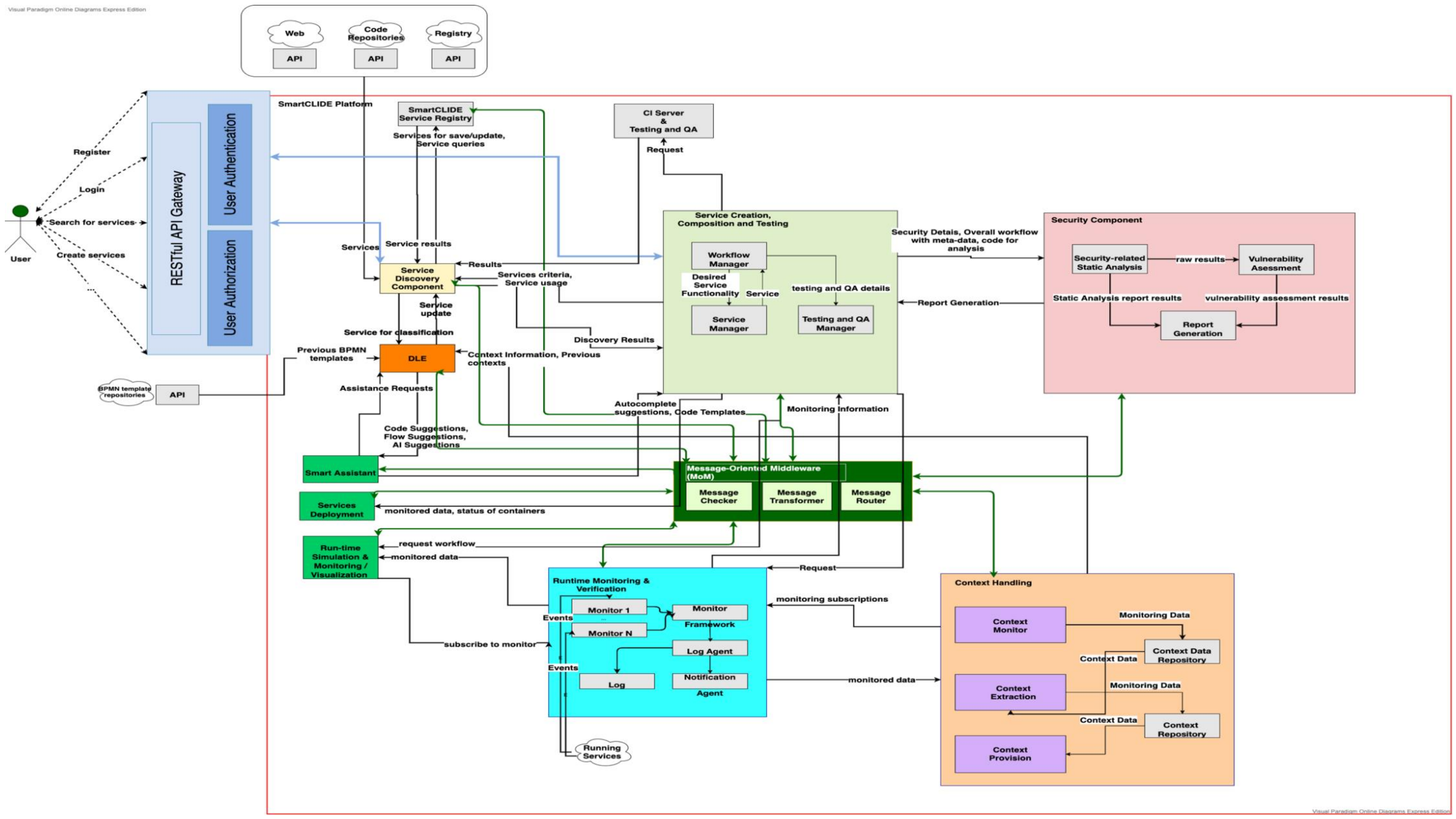


**Figure 4: SmartCLIDE Information Flows – Communication Architecture – Full Version**

The information entities handled by the system are specified in the following sub-sections, as well as the corresponding relationships among them. The descriptions of these entities are kept at a more abstract level, focusing more on the interface specifications that will be used between components during communication. Following the Consumer Driven Contracts principle, the next sub-sections describe in detail each one of the components, the invocations that take place, along with the appropriate exchange inputs / outputs.

### 4.1.1 Discovery of Services and Resources

Service Discovery is performed by an autonomous component. It takes responsibility for searching available services along with its functional and non-functional features at several heterogeneous user pre-defined sources, retrieves and hands them over to the SmartCLIDE Service Registry. Once there, they will be classified and updated by the DLE. The primary purpose is to make the services available, once classified, to the Services Composition component. Thus, services can be retrieved from three sources: Code repositories, service registries or web listings. Each of them presents some particularities regarding the obtainable information:

- Code repositories allow to reach the services code and extract the information needed to dockerize them. Code analysis (e.g. security) is performed over this information. The communication is done via API requests.
- Web services listings provide access to both REST and SOAP services, being their main disadvantage the lack of further information or the difficulty to extract it. HTTP requests will be the communication method approach to this effect.
- Service registries are a natural source for service fetching, getting functional and non-functional information from ontologies of already-existing services. The API which normally registries deliver will be the source for this kind of retrieval.

Services are stored at the SmartCLIDE service registry with all their available information. Later they are evaluated by the DLE. This evaluation consists of the classification and analysis based on their functional and non-functional parameters. It is performed regularly by the DLE, which will retrieve the unclassified services to process them utilising a predefined model[1]. This new information will be updated in SmartCLIDE service registry. Thus, services information will be queried to the SmartCLIDE service registry in case it is needed.

**Table 1: Discovery of Services and Resources cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| (External) Code repositories | To get the code from external services along with its functional and non-functional information | JSON Request | Source code files, ontologies |

[1] The classification model will be trained based on retrieved services for every source, as it requires a normalisation process for the data, an adaption to the available information and an adjustment based on the extractable categories. This last is a heuristic problem.

| | | | |
|---|---|---|---|
| (External) Service registries | To get the services from the source along with its functional and non-functional information, in the form of ontologies | JSON Request | Services information (Ontologies) |
| (External) Web service listings | To get the services from web listings along with as functional and non-functional information as possible | HTTP Request | Services JSON/XML/HTML information |
| Service Composition | (Passive, SmartCLIDE Service Registry?) Service feature-based queries | JSON Request | Service suggestions JSON |
| SmartCLIDE service repository | To save the retrieved services along with their functional and non-functional information | JSON, embedded RDF/OWL | JSON Response |

### 4.1.2 Service Creation, Composition and Testing

The Service Creation, Composition and Testing Component will interface with the Smart Assistant, Service Discovery, Continuous Integration Server and the Runtime Simulation and Monitoring Component. During the Service Creation process, the inter-component interaction is with the Smart Assistant and the Continuous Integration Server Component, during the Composition process the interaction is with the Smart Assistant and Service Discovery Component and finally during the Testing phase it is with the Runtime Simulation and Monitoring Component.

**Table 2: Service creation, Composition and Testing cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Smart Assistant | Get Code Templates<br>Get Suggestions for Workflow Autocompletion | Ontology file,<br>BPMN file | Source Code (e.g. JAVA file)<br>BPMN file |
| Service Discovery | Search for a Service | Structured Description of Functionality (e.g., XML, ontology, etc.) | JSON file (Discovery Results) |
| Runtime Simulation & Monitoring | Execute the Workflow runtime simulation and monitoring | Ontology file,<br>BPMN file (completed Workflow) | JSON file (Workflow Simulation and Monitoring data) |
| Continuous Integration Server | Register a newly created Service, to the SmartCLIDE Service Registry | Service file (service implementation),<br>Structured Description of Functionality with service meta-data (e.g., XML, ontology, etc.) | none |
| Security Component | New Service and Workflow Security and Vulnerability Assessment | Source Code (e.g. JAVA file),<br>BPMN file | Assessment |

### 4.1.3 Security

Security Component is responsible for assessing security of SmartCLIDE applications. It is comprised of three subcomponents: (i) security related static analysis, (ii) vulnerability assessment, and (iii) report generation.

***Security-related static analysis*** subcomponent will be responsible for handling all the requirements related to the identification of potentially critical security issues (i.e., potential vulnerabilities) that reside in the source code of the produced software. More specifically, this subcomponent will retrieve as input the source code of the different tasks that will be defined in the *Workflow Manager*. This source code will be retrieved from: (i) the code repository, in case a reusable code template/snippet is used, (ii) the service repository, in case that the task corresponds to the invocation of a service, or (iii) the actual code that is written by the developer, in case that no reusable service or template is available and the developer needs to create this task from scratch. The subcomponent will then invoke the execution of popular static code analysers (either open-source or commercial) in order to detect security issues that reside in the different tasks that constitute the overall workflow.

The ***Vulnerability Assessment*** subcomponent is responsible for assessing the security level of the different tasks that constitute the overall workflow, as well as of the workflow (and, in turn, the actual software) itself. In order to achieve this, this subcomponent will retrieve the source code of the different tasks and apply a set of carefully constructed Vulnerability Prediction Models. Similarly to the *Security-related Static Analysis* subcomponent, the source code will be retrieved either from the code template repository, or from the service repository. It can also retrieve the code directly from the user, in case that the task is created from scratch.

The ***Report Generation*** component will be responsible for aggregating the results produced by the *Security-related Static Analysis* and the *Vulnerability Assessment* components for facilitating their further processing and comprehension. More specifically, the raw results produced by the *Security-related Static Analysis* component will be aggregated and presented to the user in an intuitive way through Visual Analytics constructs (e.g., charts, graphs, etc.). In this section, we draft the interfaces of the "*Security*" component. The definition of the interfaces has been performed following the Consumer Driven Contracts principle and is presented in the following Table.

**Table 3: Security cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Workflow Manager | Get the overall workflow along with its meta-data (services, source code, etc.)<br>Display the results of the security analysis | JSON file | BPMN file<br>JSON file |
| Service Discovery | Get the code of a task or source code of a service | Request | Source code files |

### 4.1.4 Runtime Monitoring and Verification

The RMV requires input from the service composition process to determine which properties of the SOA application being constructed should be monitored at runtime and what values/events in the application represent the information needed to verify the properties. The RMV in response provides “virtual sensors” for identified application events for installation into the composed application. These sensors comprise calls into the Monitoring Application to register the occurrence of related events.

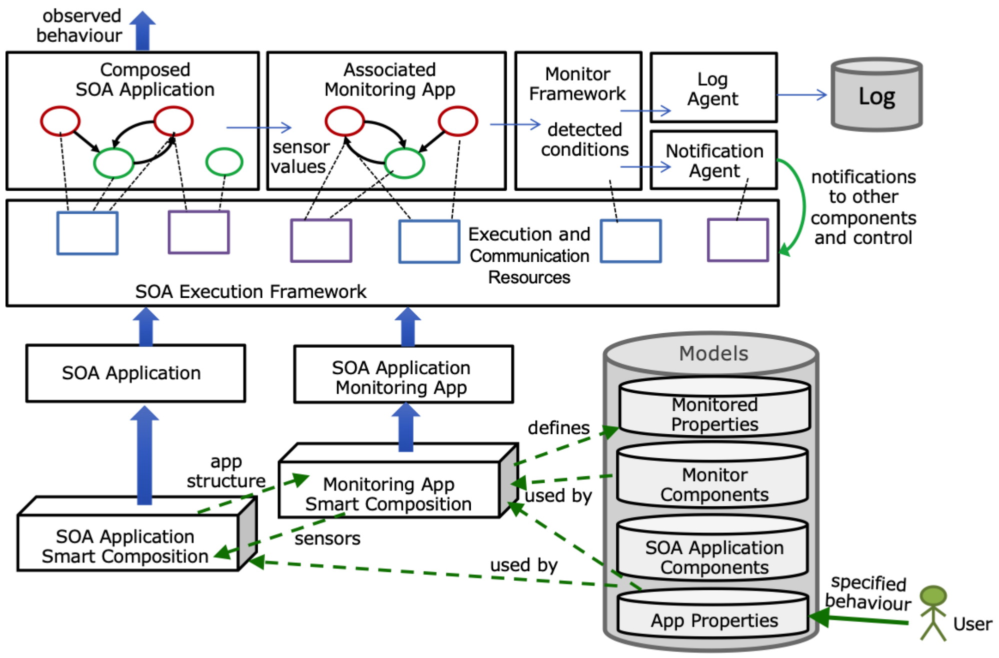


**Figure 5: Runtime Monitoring & Verification Information Flow**

The RMV requires input from the running SOA application as events. These events are consumed by the Monitoring Application specifically constructed for the specific SOA application. The event definitions are selected to enable detection of the properties of the application to be verified at runtime. There may be properties that are common to multiple applications. That is, some properties may be instantiated from a library of properties.

A Monitoring Application interfaces with a general Monitor Framework which in turn can create logs of detected conditions, verification results/failures, and generate notifications of detected conditions.

A Notification Agent will notify registered participants of conditions requiring runtime action, e.g. action to terminate or restart an application that has failed to maintain the properties that it is expected to maintain.

The Context Monitor uses the Monitor Framework to construct a monitor for events of interest and to subscribe to receive monitoring data and notifications.

**Table 4: Runtime Monitoring & Verification cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Monitoring Applications | Invoked by the monitor's corresponding application to register sensor values | Event ID | none |
| Monitor Framework | Context Monitor invokes Monitor Framework to create a monitor for context conditions of interest | Conditions to monitor | Monitor application |
| Monitor Framework | Context Monitor invokes Monitor framework / notification agent to subscribe to notifications | Where to send notifications | acknowledgment |
| Monitor Composition | Invoked by SOA application composition to create monitor | Application structure and properties | Monitoring Application |
| Notification Agent | Invoked by Monitors through the Monitor Framework to distribute notifications of detected conditions | The composed service concerned and the detected condition | none |
| Log Agent | Invoked by the monitor framework to save log records in persistent storage | Events and monitored date from Monitoring Applications | Records appended to the Log file |

The Notification Agent delivers notifications according to preconfigured or dynamically registered subscription to notifications of monitored conditions. For example, a notification would be delivered to the service manager (and possibly to deployment and the service execution framework) when a monitor's triggering condition is met, such as a failure of a service to behave according to its specified properties.

The Log Agent stores log messages to a persistent store and manages that store. Log messages may correspond to events generated within the monitoring framework or to events reported by other applications or SmartCLIDE infrastructure.

### 4.1.5 Runtime Simulation & Monitoring / Visualisation

The Runtime Simulation & Monitoring / Visualisation Component represents the front-end for the Monitoring Module in SmartCLIDE. For this reason, this component interfaces with the Runtime Monitoring and Verification Component, that provides a framework and monitoring data of any of the applications (containers) already running in SmartCLIDE. Once the monitoring is gathered, the Runtime Simulation & Monitoring / Visualisation Component represents it through a visual console, allowing the developer to track the status and/or performance of deployed instances.

**Table 5: Runtime Simulation & Monitoring / Visualisation cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Runtime Monitoring & Verification | Get list of available monitors. | JSON Request | JSON Response |
| Runtime Monitoring & Verification | Get value of a given monitor. | JSON Request | JSON Response |
| Runtime Monitoring & Verification | Subscribe to a given monitor. | JSON Request | JSON Response (periodical) |
| Runtime Monitoring & Verification | Unsubscribe to a given monitor. | JSON Request | JSON Response |

### 4.1.6 Deep Learning

The Deep Learning Engine is an autonomous component which has the following functionalities:

- Classifies services from several sources based on their functional features, users' feedback or prior usages.
- Identifies the context delivered by context monitoring component in the form of ontologies or JSON files.
- Enables abstraction selection, or the assistance to the composition of services while BPMN workflow modelling.
- Generates or assists to the generation, of programmatic output in the form of plain code, code templates or AI models.
- Gives support to functionalities delivered by the Smart Assistant which need an intelligent behaviour.

Services information will be retrieved and updated from/at the SmartCLIDE services repository. These services have previously been fetched by the Services Discovery component taking user-defined sources as origin.

Code generation feature will be trained with existing code for specific functionalities (i.e. database connections, credentials checking, etc.) to deliver quality code templates whose parameters can be autocompleted. This reference code will be extracted from the consortium repositories via API.

The abstraction selection model will be trained with existing repositories of previously used BPMN templates. The suggestions over the next suitable elements in BPMN workflows will be provided in a JSON with the required information.

Context Identification will get the information from the Context Monitor in the form of JSON or RDF ontologies, to train the AI models and deliver its results.

Concerning the assistance to generate AI models, the user's input in the form of structured files will be processed to extract the relevant fields and generate a model. These models will be provided as a ZIP file with an embedded API to interact with them.

The Smart Assistant AI-based suggestions will need the following input data in a structured mode, to enable the AI selected algorithm to create a model or be able to use it.

**Table 6: Deep Learning engine cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| User input | (Passive) To generate or utilise AI models | Data Files | JSON response, Zipped/dockerized models |
| Context Monitor | To get abstractions for what the current context is | Abstractions | RDF/OWL, JSON |
| BPMN repository | To get previously used templates to learn from them and provide next-element suggestions in composition workflow | JSON Request | BPMN templates |
| Services Creation | To get code templates | JSON Request | JSON, embedded template code |
| SmartCLIDE service repository | To get the source code or the features of a service | JSON Request | JSON, source code + ontologies |
| SmartCLIDE service repository (II) | Get unrated services, update already-rated services | JSON Request | JSON Response |
| Smart Assistant | (Passive) Suggestion queries based on current user activity or other component's outputs | JSON Request | JSON Response |

## 4.1.7 Context Handling

The Context Handling Component will interface with the Runtime Monitoring and Verification Component to receive monitored data.

It is foreseen that monitored data can be received both in a pull and push fashion from the Runtime Monitoring and Verification Component. To pull monitored data for a specific monitor and sensor the Runtime Monitoring and Verification Component shall provide a corresponding Rest-API - see "Get the monitored values for a given monitor and sensor" in Table 7.

To support pushing of monitored data into the Context Handling Component the Runtime Monitoring and Verification Component shall provide a subscription mechanism in form of a Rest-API where the Context Handling Component can register itself as a subscriber for the monitored data of a specific monitor and sensor - see Table 7. Based on the existing subscriptions, the Run-time Monitoring and Verification Component will publish monitored data via the Message Oriented Middleware.

The Context Handling Component in turn will provide the necessary interface to receive monitored data, either as a Rest-API or via the Message Oriented Middleware , depending on the amount of monitored data and frequency.

**Table 7: Context Handling cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Runtime Simulation and Monitoring | Get a list of all available monitors | none<br><br>HTTP query parameters to filter and sort list of monitors | JSON array of available monitors |
| Runtime Simulation and Monitoring | Get the monitored values for a given monitor and sensor | HTTP path parameters “monitorID” and “sensorID” | JSON array of monitored data |
| Runtime Simulation and Monitoring | Subscribe to a given monitor and a given sensor | HTTP path parameters “monitorID” and “sensorID”<br><br>JSON data representing the “address“ where to send monitored data | JSON object representing a subscription |
| Runtime Simulation and Monitoring | Unsubscribe from the given subscription | HTTP path parameters “monitorID”, ”sensorID”, and “subscriptionID” | JSON object representing Success or Failure of operation |
| Deep Learning | Send current identified context. | RDF model | JSON object representing Success or Failure of operation |

Additionally, the Context Handling Component may provide a Rest-API endpoint, which returns an RDF model of the current identified context.

### 4.1.8 Smart Assistants

This component takes the user's actions within the interface in the form of events and returns suggestions through the IDE interface, all depending on the current software lifecycle stage. Thus, it is dependent on the IDE events API. Once the events are thrown and provided the current software development stage, either a DLE algorithm or a third-party plugin will be queried for a suitable suggestion given the user context.

It either makes queries to the DLE or a specific plugin depending on the nature of the query.

Thus, the source of information, depending on the desired assistance and development stage, can be:

- Third-party modules, called in the event of an already-developed assistance functionality.
- The DLE, which will provide input mechanisms for structured data to (i) train the functionality model; and (ii) make use of it.

As mentioned above, it will be needed to determine the current context of the user:

- During the Requirements & Design stage, inputs will be:
  - The Gherkin schemes and functionality provided by the user
  - The current BPMN workflows along with user inputs on the IDE interface
- During the Development stage:
  - The programmer input via the IDE interface
- During the Testing stage:
  - The test results
- During the Deployment stage:
  - The Service Deployment component. Consortium components information will be available all through the platform via an API.

All these pieces of information will be retrieved from the matching component through its API when the matching interface event is triggered.

The SmartCLIDE assistant capabilities and matching plugins, as well as the assistance feasibility regardless of the mock-ups provided in D1.4, will be tested during the project development, being the AI model creator assistant the initial reference to this purpose.

**Table 8: Smart Assistants cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Deep Learning Engine | To get suggestions that match the current user status or action | JSON Request | JSON suggestions |
| Services Composition | (Passive) To return suggestions on next suitable items for BPMN flows | JSON Request | JSON suggestions |
| Theia Interface | (Passive, events) To get what the user is doing at each stage of software lifecycle. | Theia interface events | JSON queries (to other components) |
| Other components | To get other components’ outputs for analysing and returning suggestions at a specific software lifecycle stage | JSON Request | JSON Response |

### 4.1.9 Services Deployment

The Services Deployment component represents the front-end for deployment tools. As defined in D1.4, this component can be used by the developer by using a Command Line Interface, or a Web-based application that not only allows the developer to deploy a new service (or a complex system built from more than one service), but also to check the status and health information about existing containers.

To achieve this behaviour, Services Deployment component retrieves the list of existing containers or images, and their status (e.g., health, network configuration…) from Workflow Manager which interacts with the Runtime Monitoring and Verification component. Moreover, it also needs to access any needed backend docker commands (e.g., docker pull, docker run…) to use its engine to deploy, manage or stop containers.

**Table 9: Services Deployment cross-component interfaces.**

| Component | Goal of Invocation | Input | Output |
|---|---|---|---|
| Workflow Manager (Services Creation, Composition and Testing) | Retrieve information about existing containers (e.g., status, health…), available docker images... | JSON Request | JSON Response |
| Services Creation, Composition and Testing | Deploy a new container using back-end commands (e.g., docker) | JSON Request | JSON Response (result) + new container |

### 4.1.10 API Gateway

CERTH will implement and design a **RESTful API gateway** through which all functionalities of the SmartCLIDE platform (from user registration to service discovery and service creation) will be exposed to the user. This component will also provide **authentication and authorization functionalities** using the best industry standards like **OAuth2** [1]. Several technologies can be used to implement this component. CERTH has extensive experience in designing and developing similar gateways using **Python technologies like Flask** [2] and **Eve** [3].

### 4.1.11 Message Oriented Middleware

This component will be responsible for the inter-component communication within the SmartCLIDE platform. It will be implemented as a **message broker** and will provide **asynchronous** communication functionalities based on the **publish-subscribe** (i.e. pub/sub) pattern. The component will facilitate the **loose coupling** of the components, which will enhance their **developability** and **maintainability**. The pub/sub pattern is capable of serving **multiple senders and multiple receivers simultaneously,** and therefore scalability will be inherently supported. The component will provide **message routing**, **transformation,** and **validation** functionalities. In addition, specific **security policies** will be implemented within the component. There are several message broker’s software available, with popular choices being the **Apache Kafka**[6], **Apache Qpid**[7].

# 5 System Deployment View

This section presents the deployment view of the core components of the SmartCLIDE platform, as shown in the following figure, paving the way towards the successful system integration. It connects the Frontend and Backend services of the platform as well as the pilot sites infrastructure located either on Public or Private cloud.

At a high-level view, the SmartCLIDE platform consists of following major components:

- the SmartCLIDE IDE
- the RESTful API Gateway
- the Backend System
- the Message Oriented Middleware
- the Deployment infrastructure

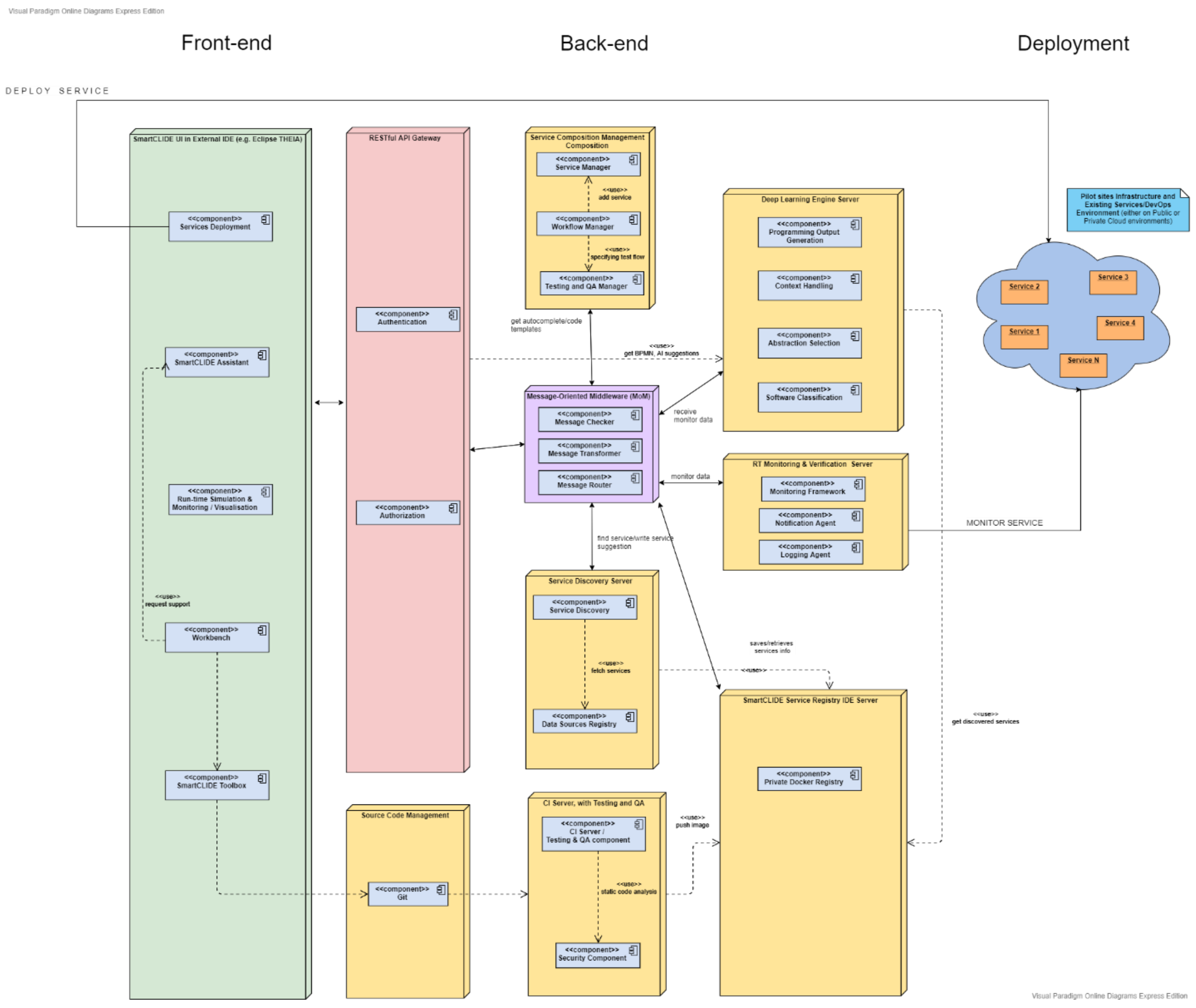
The SmartCLIDE IDE is a cloud-based software development environment that includes the SmartCLIDE Toolbox, the Assistant, the Runtime Simulation and monitoring/visualisation and the Workbench. The Toolbox supports the software development lifecycle and provides all the necessary tools to specify new features, acceptance tests, unit tests and integration tests. The users may also define a BPMN Workflow through the Workbench and visualize its state with the Runtime Simulation and monitoring/visualisation component. Additionally, the Assistant helps both technical and non-technical users with the software development lifecycle by providing recommendations and real-time suggestions.

The RESTful API Gateway addresses all the authentication and authorization concerns regarding the use of the Backend services. Authentication deals with whether a Client is allowed to connect to a specific server providing the API while authorization is concerned with whether a particular Client after connection is allowed to perform the task in question or otherwise have access to a specific resource. The Basic authentication approach is the simplest one where an HTTP header with username and password information is included in base64 encoding. Due to the fact that this approach contains the credentials unencrypted it should only be used with encrypted SSL/TLS connections. To avoid the overhead of sending the credentials with each request and avoid tampering of the headers or body of the request, Hash Based Message Authentication (HMAC) represents an option where the Client sends a digest of the request and a nonce that based on a shared secret between Client and Server it can be verified.  A more common approach used is the OAuth 2.0 token-based authentication where Clients after registering with a provider are given a token to be used with every request. Requests that have been altered in any way or contain an invalid token are rejected. In general requests should be made over SSL/TLS otherwise the token can be stolen. Tokens can also be made to expire after a certain period or be revoked to restrict access when necessary.

The Backend System is a multi-functional component including many other sub-components such as the Service Creation and Composition management, the Deep Learning Engine, the Service Discovery, the SmartCLIDE IDE Data sources and the CICD related components Git, CI server and SmartCLIDE Service Registry. It interacts with both the frontend SmartCLIDE IDE to provide the necessary support to its functionality and the Deployment environment to produce the user application services. These services are deployed either on Public or Private Cloud according to the user needs.

It has been identified from the early stages of the project that the main focus was on enabling to the greatest extent and in the best possible way a scalable, fault-tolerant communication-efficient framework. For this purpose, a Message Oriented Middleware has been added to serve as the communication layer among different components of the Backend System.

In terms of hosting, all the microservices developed by the end users is recommended to be hosted in either public or private cloud virtual machines infrastructure with a Unix-based OS. In case of any project restriction or different approach selected for any purpose, compliance with any other hosting model will be feasible.



Visual Paradigm Online Diagrams Express Edition
Front-end
Back-end
Deployment
DEPLOY SERVICE
SmartCLIDE UI in External IDE (e.g. Eclipse THEIA)
<<component>> Services Deployment
<<component>> SmartCLIDE Assistant
<<component>> Run-time Simulation & Monitoring / Visualisation
<<use>> request support
<<component>> Workbench
<<component>> SmartCLIDE Toolbox
RESTful API Gateway
<<component>> Authentication
<<component>> Authorization
Service Composition Management Composition
<<component>> Service Manager
<<use>> add service
<<component>> Workflow Manager
<<use>> specifying test flow
<<component>> Testing and QA Manager
get autocomplete/code templates
<<use>> get BPMN, AI suggestions
Message-Oriented Middleware (MoM)
<<component>> Message Checker
<<component>> Message Transformer
<<component>> Message Router
receive monitor data
monitor data
find service/write service suggestion
Deep Learning Engine Server
<<component>> Programming Output Generation
<<component>> Context Handling
<<component>> Abstraction Selection
<<component>> Software Classification
RT Monitoring & Verification Server
<<component>> Monitoring Framework
<<component>> Notification Agent
<<component>> Logging Agent
MONITOR SERVICE
Pilot sites Infrastructure and Existing Services/DevOps Environment (either on Public or Private Cloud environments)
Service 2
Service 3
Service 1
Service 4
Service N
Service Discovery Server
<<component>> Service Discovery
<<use>> fetch services
<<component>> Data Sources Registry
saves/retrieves services info
<<use>>
SmartCLIDE Service Registry IDE Server
<<component>> Private Docker Registry
<<use>> get discovered services
Source Code Management
<<component>> Git
CI Server, with Testing and QA
<<component>> CI Server / Testing & QA component
<<use>> static code analysis
<<component>> Security Component
<<use>> push image
Visual Paradigm Online Diagrams Express Edition

**Figure 6: SmartCLIDE Deployment Architecture**

The components that will use the MoM can be briefly described in Table 10.

**Table 10: Components using MoM.**

| Component | Puspose |
|---|---|
| RT Monitoring and Verification Server | Exchange monitoring data |
| Deep Learning Engine | Context Handling sub-component will communicate with the RT Monitoring and Verification Server |
| SmartCLIDE Service Registry | Get services |
| Service Discovery | Write services suggestions |
| Service Composition Management | Retrieve autocomplete/code templates |

## 5.1 Deployment at Pilots

### 5.1.1 PERSEUS Pilot – INTRASOFT International

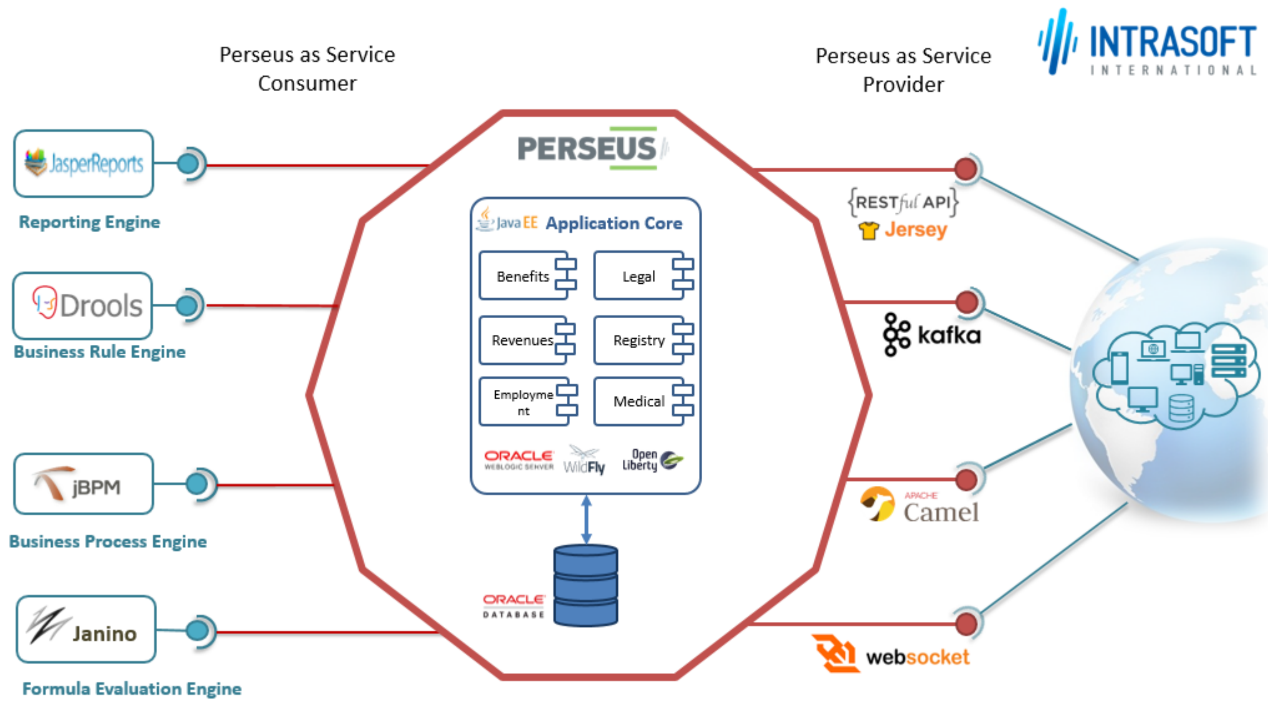


**Figure 7: PERSEUS Current Deployment View**

Perseus product is based on a multitude of state-of-the-art technologies which are all combined in a decoupled and scalable setup. The core business application logic exists in a number of Java EE Web applications (Figure displays only 6 for clarity) deployed in a cluster of Java EE Application servers. Business logic is externalized from the core application logic by using external Business Process and Rule Management Engines which are connected through Adapter APIs with the core application. On the other hand, Perseus provides numerous outbound integration endpoints such as Rest API, WebSockets and also Kafka and Camel generic adapters.

**Deployment**

A crucial part of complex software systems development and delivery lifecycle is Continuous Integration (CI) and Continuous Delivery/Deployment (CD) – jointly mentioned as CI/CD. CI, in software development, is a practice of building/integrating and testing all developer working code frequently in a shared code repository. A common practice in CI is to integrate the changed code at least daily. The frequent integration helps the contributors to notice any arising errors and correct them instantly.

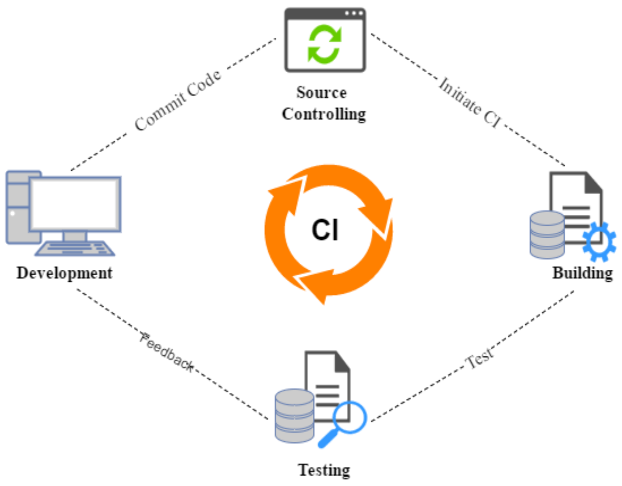


**Figure 8: The Continuous Integration Lifecycle**

**Continuous Integration** offers significant advantages such as:

- reduction of risk in the development, since it reveals any possible incompatibility between software components early during the development phase
- facility of instant bug fixing
- availability of current product version at any moment, as the code is frequently integrated

In order to enable the CI process and benefit from the aforementioned advantages we have to perform extensive testing of each software component/module but also of the integrated platform as a whole. Automated per component tests and combined integration tests that are performed on each new build (version) of a component shall be executed and in case of success the integrated platform will be updated with the new version of the component. Otherwise the developers will be notified so as to take proper action to fix the problem that caused the failure. A Test-Driven Development (TDD) approach may be followed, starting from unit tests per software component, upon specified test cases for each component, proceeding to integration tests that validate the correct functionality of the integrated platform that involves two or more components in an automated manner with the use of a CI server, such as Jenkins[2]. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

[2] https://jenkins.io/

In case that the resulting software system is composed of several software components/modules, developed by diverse development teams, a collaborative software development approach can be followed, during which it is important to ensure data integrity and availability. To support this, a distributed version control system (VCS) is necessary for the efficient system software delivery. For this purpose, GitLab[3] could be utilized, a source code management and VCS that aims mostly at data integrity and full version tracking. GitLab is an easy yet powerful and intuitive git VCS. Multiple developers can concurrently create, merge and delete parts of the code they are working on independently, at their local system before applying the changes to the shared GitLab repository. A DevOps framework could be adopted.

To ensure quality of software development, build automation is additionally pursued. Build automation is considered to be the act of automating processes that are associated with software building. Such processes might include various parts like source code compiling into binary code, packaging binary code and automated test running but also the delivery/deployment and documentation parts. Maven[4] is a useful tool for build automation and project management for projects that are written in numerous programming languages (Java, Ruby, C# and other). This process can be applied for components software shared and residing in the software source code repository.

It is also essential in collaborative software development to distribute the software efficiently. Among others, Nexus[5] is a popular repository manager that manages the required software artifacts. It allows developers to distribute their software easily but also to proxy, publish and collect the necessary project dependencies. Actually, the Nexus Repository offers a standardized way for cataloguing and storing developers' artifacts. Once a new library is developed, it is handed out to the repository manager. After that, other developers can efficiently access these software components by using a standardized procedure. Clearly, it is possible to control centrally the development of all artifacts and the access to them. Nexus can be used for pre-built software components, the source code of which is not available or shared – however, updated versions of software components need to be frequently built and released in new versions at the Nexus repository to support continuous integration and delivery. The CI server will monitor the repository and will initiate a new platform test and deployment cycle after each new version upload.

To further support automated delivery (**Continuous Delivery - CD**) in software development, Docker[6] can be chosen as a straightforward way to provide isolated running environments with pre-set configuration. Docker is an open-source software containerization platform and it works with software containers in order to allow the software to run always the same, independently of the deployment environment. Actually, it wraps up the software in a complete file system, along with any necessary tools or software resources, such as libraries, code and runtime. This way, multiple docker containers can run on a single Linux instance, without any overhead for

[3] https://about.gitlab.com/
[4] https://maven.apache.org/
[5] https://www.sonatype.com/product-nexus-repository
[6] https://www.docker.com/

managing several virtual machines. Moreover, by using Docker, the software system deployment is simplified at a great extent, set up is minimal and uniform across all component projects. Each component is contained in a different Docker image ready to be executed in a Docker hosting machine as a separate container. These images can be published in a shared repository, such as the Docker registry, and through the Docker Compose functionality these images can be retrieved from the Docker registry and deployed together via a single configuration file. Containerization thus provides OS level virtualization. This means that multiple applications running in containers on a single host, access the same OS kernel. Hence, it is faster and more lightweight than isolating applications using VMs. Containers have an initial configuration which does not affect the configuration of other containers, even though they share the same host OS. This eliminates errors due to unexpected conflicts or missing dependencies, which are common when applications are installed on a single host without isolation. In more demanding installations due to increased load of the system, Docker is perfectly suitable to be configured with load balancing mechanisms that can scale up the performance of the system.

Kubernetes[7], an open-source system created by Google, has gained popularity over the past few years and has become the industry standard for deploying containers in production. It allows the deployment, management, and monitoring of multi-container applications at scale. These functionalities along with the use of pods, clustering, rolling updates, load balancing and horizontal scaling results in dynamic scaling that can be achieved for a specific element, microservice or container. Moreover, by using deployments, it provides an easy way to scale and update pods always online. Kubernetes is the most widely adopted orchestration tool that has been integrated due to its open-sourced nature to many corporate-ready cloud solutions.

The following figure depicts the CI/CD workflow with all the tools mentioned above, and summarized below:

- GitLab for source control, acting as code repository and allowing code versioning
- Jenkins for automated build and testing
- Docker for containerization of services and components
- Docker Registry for easy deployment at different infrastructures

[7] https://kubernetes.io/

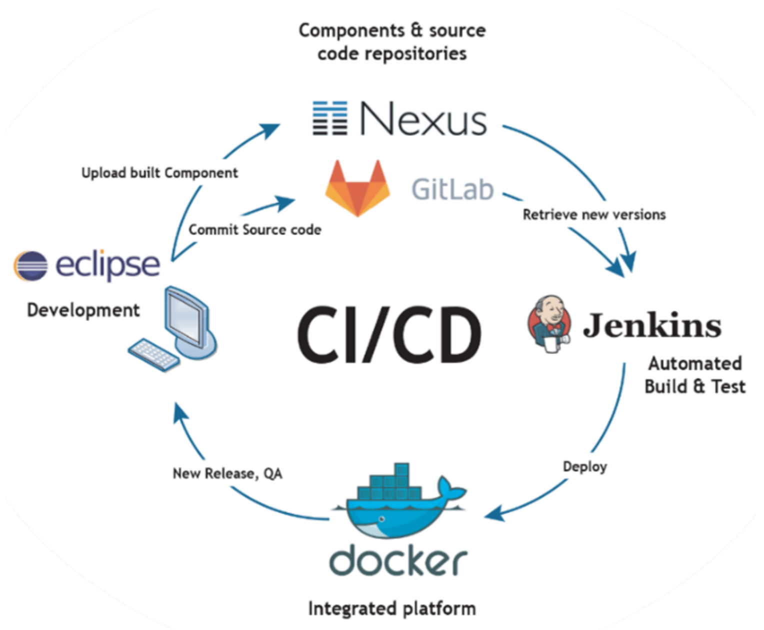


**Figure 9: Continuous Integration & Continuous Delivery process**

Using the CI/CD environment and tools, when developers implement new component features or integration endpoints, they *push* their code to GitLab, the central source code repository in this environment. The code is then compiled, built and tested using Jenkins. Finally, Docker is creating a Docker image, that is pushed to the Docker Registry.

Once components have been built and their images have been pushed to the docker registry, they are available to be pulled from *any server* which has access to the docker registry. The deployment to Deployment servers can be carried out via a script which automates the entire process.

SmartCLIDE should thus provide the user with the following capabilities as described in detail in D1.2: Requirements Analysis:

- Provide a complete, easy-to-use, configurable online integration platform that will enable business process design in a drag-and-drop nature, but also supporting low-code capabilities.
- Following the previous action Real-time compilation, verification and deployment of the developed scenarios should take place.
- SmartCLIDE should offer a common adapting layer where all services deployed will communicate with the external world in the customer-desired formats, but use a single common, homogeneous API for internal SmartCLIDE usage.
- Easy integration and deployment to a running system, with minimum downtime and zero actual application re-deployments where possible.

### 5.1.2 Real-time Communication Platform Pilot – Wellness Telecom SL

Real-time communication platforms have become a key service in businesses. In fact, running them on the cloud is a great way to manage their own privacy and requirements. Cloud-based real-time communication platforms are commonly named UCaaS – Unified Communication as a Service –, and are typically composed by several elements or entities (i.e., microservices) that conform the global service. Using containers is a great way for providing both security and flexibility, getting advantage from its inherent characteristics (e.g., network isolation, or replication). In fact, real-time communication services have strong QoS requirements, which in turn depends on the performance of network or any involved systems (i.e., CPU resources), so that real-time adaptation is a very interesting characteristic in this kind of services.

Next, we elaborate on the entities that conform the whole system in Section 5.1.2.1. Then, in Section 5.1.2.2 we show how each entity collaborates with the rest in real use cases (from end-users' point of view). Finally, we introduce the expected implementation in 5.1.2.3.

#### 5.1.2.1 Entities in the system

The overall system can be seen as a unified system from the end-user point of view as illustrated in the following figure. In this figure, each user accesses the service using a SIP agent (e.g., browser, softphone…) which assumes two types of information flows: signalling (e.g., negotiation, call establishment, hang) and voice/video data. Observe that data may be (de)multiplexed during the exchange process given the fact that one-to-many and many-to-one communications are allowed (e.g., conference rooms). On the other hand, note that there is an entity named "system administrator" that has direct access to the platform (e.g., ssh), for example to assess maintenance tasks.

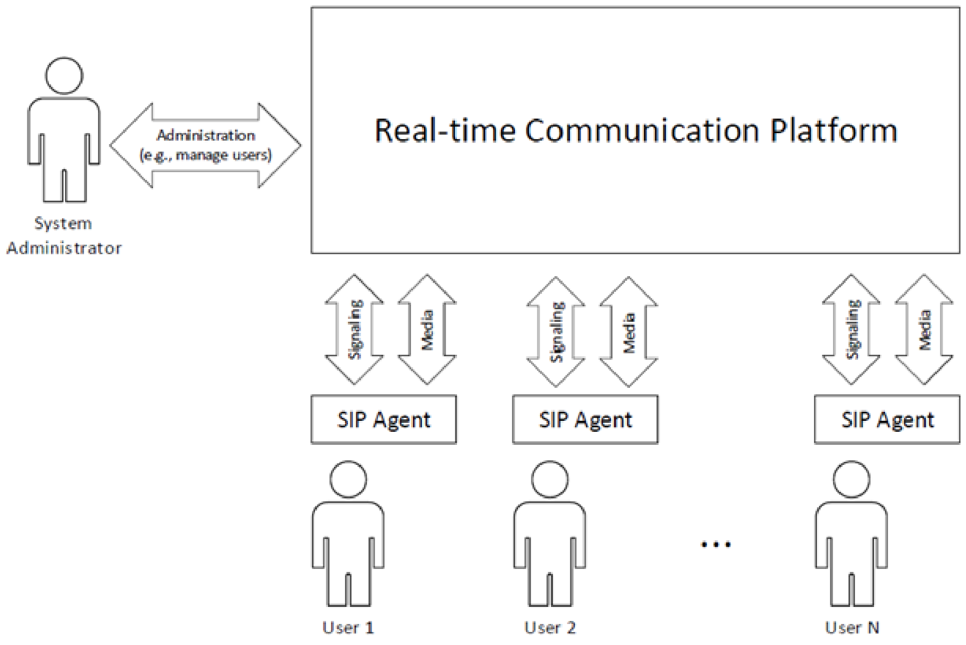

Real-time Communication Platform
Administration (e.g., manage users)
System Administrator
Signaling
Media
SIP Agent
User 1
User 2
...
User N

**Figure 10: Real-time Communication Platform**

Inside of the communication platform, there might be a variable number of entities depending of the offered service and its capabilities. In WT's case, the collection of entities in the system is defined in the following table.

**Table 11: Collection of the system entities**

| **Entity** | **Usage** |
|---|---|
| Voice gateway | Gateway and balancing for VoIP (signalling) |
| Video gateway | Gateway and balancing for VideoIP (signalling) |
| Web server | Allows users to join or start a call using a browser (i.e., embedded SIP Agent) |
| RTP Proxy | Redirects RTP connections to the corresponding server |
| VoIP server | Handles SIP signalling (e.g., call establishment) for audio calls |
| VideoIP server | Handles SIP signalling (e.g., call establishment) for video calls |
| Mixer and/or transcoder | Provides transcoding and mixing video tools |
| Database | Provides storage capabilities (e.g., registered users, calls in progress) |

Each of these entities can be represented as illustrated in the following figure. In the figure, each entity is represented as an independent instance that communicates with each other using an internal network named "overlay network". In order to provide the communication service to the end user, some of the entities have also to expose a series of ports over the external network (i.e., LAN). Users must have access to the following entities:

- The web server, to log into the service and join or create any existing call. This component also acts as a SIP agent (i.e., embedded), allowing users to trigger calls without using external applications.
- The gateway, to send and receive any signalling data (e.g., call establishment, hang…)
- The RTP proxy, to send and receive voice or video flows.

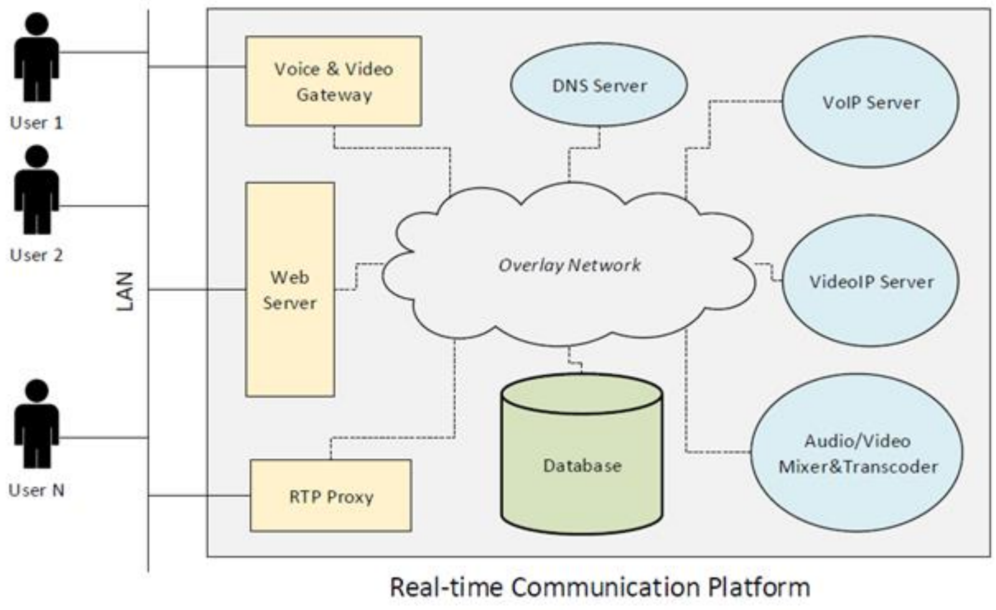


**Figure 11: Real-time Communication Platform Components**

### 5.1.2.2 Usage workflow

In order to understand how each element in the previous figure may collaborate, let us consider a brief example of a simple peer to peer call. Let's consider that "User 1" is the caller, and "User 2" is the callee. The following steps are completed until the call is established:

1. The caller logs into the web-based application, which includes a SIP Agent and triggers the call.
2. The SIP agent will access to the gateway, using the LAN and through port:5060. This request will use SIP protocol for signalization.
3. The gateway will check in the database that the user who starts the call is registered in it. If yes, the gateway service will check which VoIP server has to handle the call.
4. The gateway will forward the call to the chosen VoIP server using SIP.
5. The VoIP server will check the database to know some information related to the callee. This information is necessary to establish communication with the receiver.
6. Once this information is retrieved, the server will answer back through the gateway (to continue the signalling handshake).
7. The gateway service will use SIP protocol to communicate with the callee's SIP agent.
8. This SIP agent will ring waiting for the receiver's acceptance.
9. Once it is verified the communication is possible, the gateway service communicates with RTP proxy to select free ports to handle the data streams for audio properly, solving problems related to NAT.

10. From this moment on, the RTP will manage directly data flows from both users, supporting the traffic between two users

### 5.1.2.3 Deployment

In a cloud environment, each of the previous entities may be deployed as one or more containerized instances depending of replication or scalability requirements. Find below a table that maps each entity with its containerized application.

**Table 12: Network - scalability requirements**

| Name | Functionality | Network exposure | Required instances (#) |
|---|---|---|---|
| Kamailio | Voice gateway | Yes | 1 |
| | Video gateway | | |
| Web portal | Web server | Yes | 1 |
| RTP Engine | RTP proxy | Yes | At least one |
| Doubango | VoIP server | No | At least one |
| | VideoIP server | | |
| | Mixer and/or transcoder | | |
| MySQL | Database | No | 1 |
| Mongo DB | | No | 1 |
| Redis | | No | 1 |

A brief description of each container can be found as follows:

- **Kamailio**: This is a very flexible open source SIP proxy server, which can be used to build complex real-time communication platforms; it can also support Presence and Instant Messaging functions. This service can also act as a load balancer of VoIP and video calls along with other servers. In the present use case, Doubango servers are used as video conference servers while Kamailio is responsible for the signalization protocol.
- **MySQL**: An open-source management system for relational databases based on Structured Query Language. In this case, a MySQL database is used to store registered users in the system, as well as the calls registry and another dataset important for the proper operation.
- **RTP Engine**: This service allows media data flows to circulate through itself supported by the RTP protocol. The data flows can be both audio and video flows.
- **Doubango**: This service is an open-source SIP TelePresence System. It is used as a video mixer for shaping various streams at conferences.

- **Web Portal**: This service exposes a website that offers access to the communications system without installing additional software in user devices. This portal provides an agent (SIP agent) which can be executed in the browser and lets anybody registered in the system use the service without having any other program installed in his device.
- **Mongo DB**: This is an open-source cross-platform document-oriented database program, and is used as a database for the web portal.
- **Redis**: an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It is also used as a database for the web portal.

Finally, a monitoring agent may be considered to collect monitoring data within the internal network, and to push it to the SmartCLIDE monitoring server. Figure 12 provides a snapshot of the final architecture of the components of this pilot use case.

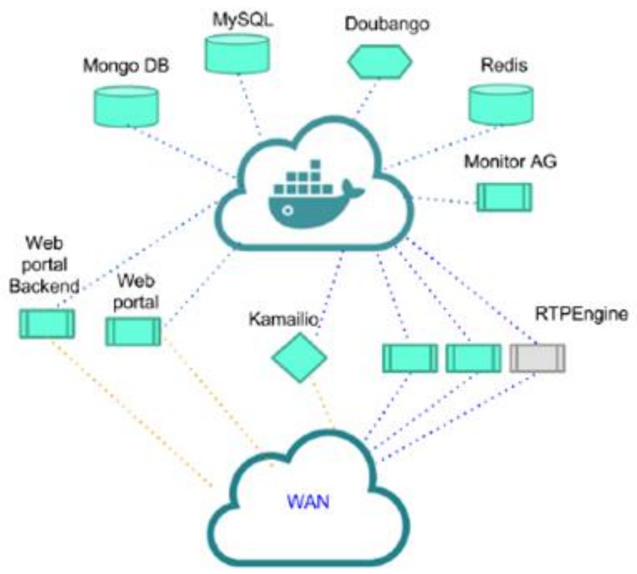


**Figure 12: Real-time Communication Platform Architecture**

SmartCLIDE should allow the developer to deploy the described pilot case by following the next steps:

1. Developer will explore SmartCLIDE repository and find the real-time communication platform.
2. Developer will load the real-time communication platform in the IDE.
3. Developer will be able to customize the deployment (number of instances – replicas of RTP Engine or Duobango containers, ports exposed to external network…).
4. Developer will be able to deploy the service.

During the service execution, SmartCLIDE should assist the developer in:

a. Management tasks, allowing the developer (or IT manager) to connect to any of the deployed instances (e.g., ssh to MySQL).

b. Re-deploying the service, e.g., to increase the number of instances of RTP Engine or Duobango dockers.
c. Monitoring any of the existing dockers.
d. Programming automatic scaling tasks, such as replicating RTP Engine instances whenever is needed (e.g., the number of open ports is exhausted).

### 5.1.3 IoT-Catalogue IDE – UNPARALLEL Innovation, LDA

The IoT-Catalogue (https://www.iot-catalogue.com) is a web-based tool that allows IoT stakeholders to explore innovations in IoT applications and technologies. It is a wide repository of knowledge, use cases, contacts, etc. of the Internet of Things that allow users to pick & choose IoT solutions.

#### 5.1.3.1 Current IoT-Catalogue Infrastructure

The IoT-Catalogue infrastructure is composed by several components deployed on different machines and on different geographical spaces. Components are distributed across different Docker stacks, being one stack deployed on Amazon Web Services (AWS) while others are deployed in servers at UNPARALLEL's premises.

This section describes the current setup of IoT-Catalogue infrastructure. Figure 13 presents a diagram of all components included in the current architecture. Each of the component is described, explaining its function. Also, it is explained how each component is connected.

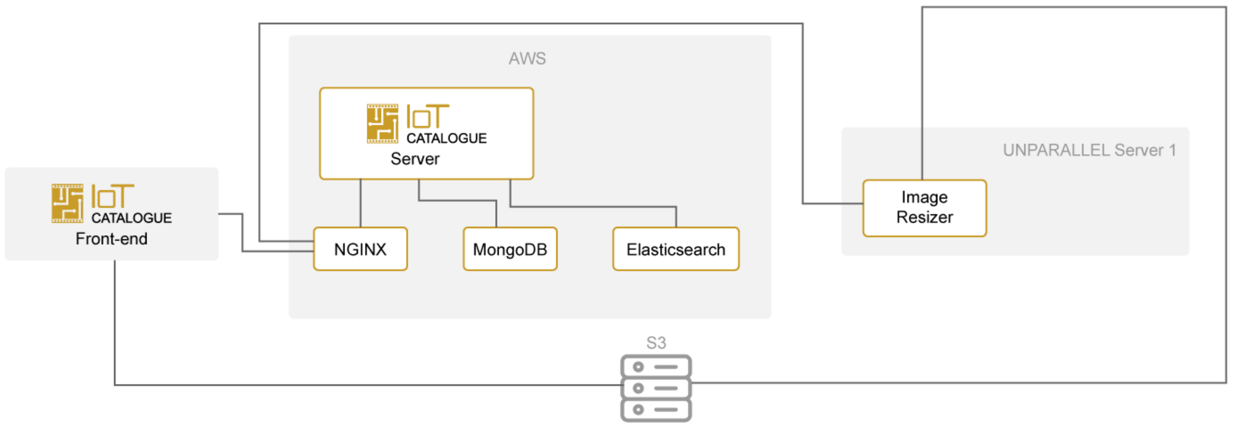


**Figure 13 - Overview of the current deployment of IoT-Catalogue components**

- **IoT-Catalogue Front-End**: IoT-Catalogue, being meteor Framework based, it has its code split between server and client, with the communication established via WebSockets. The Front-End represents the part that runs on the client's browser.
- **IoT-Catalogue Server**: The Server section, is which enforces all the control of the information, access to the database, user sessions, etc. and handles all the user requests.

- **Nginx**: All the connection to outside world is tunnelled by the Nginx, this establishes secure channels to the users (SSL), and routes the requests to the different services.
- **MongoDB**: Database to store all the assets of the platform.
- **Elasticsearch**: Indexing and relate information is one key aspects of the IoT-Catalogue, Elasticsearch is used as a support tool for providing this.
- **S3**: All the static content of the IoT-Catalogue is stored in a cloud storage service, such as Amazon S3.
- **Image Resizer**: IoT-Catalogue rely on heavy processing services, such as this service to resize images to web friendly sizes. This, and other services like this, run locally in UNPARALLEL premisses, on local infrastructure, and connects to the IoT-Catalogue with two mechanisms. One WebSocket, to receive real time updates on information change, such as being informed when new images are uploaded. And a REST API, for publishing new information on IoT-Catalogue. Additionally, these services can also connect to external services such as Amazon S3 to upload images.

The next section explains how SmartCLIDE will improve the IoT-Catalogue infrastructure.

#### 5.1.3.2 SmartCLIDE empowered IoT-Catalogue Infrastructure

An opportunity was discovered. IoT-Catalogue needs a way to provide to the community integrated development capabilities. SmartCLIDE will provide such capabilities, by providing an IDE. This IDE will provide the ability to develop inside the IoT-Catalogue in an integrated approach. SmartCLIDE’s other contribution is based on enriching the current information that IoT-Catalogue, which will be done by the indexation and classification of the services included in IoT-Catalogue. By adding this ability, the current infrastructure will need improvements and updates.

Figure 14 shows a possible architecture, resulting from the integration of the current infrastructure of IoT-Catalogue with the SmartCLIDE.

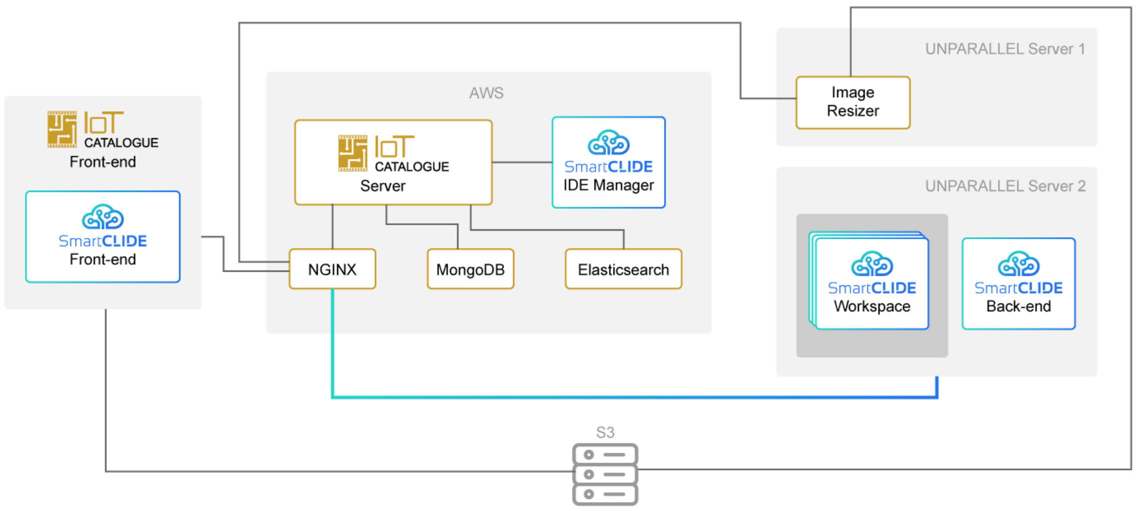


**Figure 14 - Overview a future deployment with SmartCLIDE components integrated in IoT-Catalogue infrastructure**

Such improvements need to work over the current IoT-Catalogue architecture, and are as follows:

- **SmartCLIDE Front-End**: For a seamless integration in IoT-Catalogue, the SmartCLIDE IDE is desired to be presented in a JavaScript library, preferably in React.js.
- **SmartCLIDE IDE Manager**: For optimization purposes, and to facilitate the Authentication procedures, a coordination module should be deployed on the same stack as the IoT-Catalogue server and database for security purposes.
- **SmartCLIDE Workspaces**: Since SmartCLIDE will rely on one workspace per user, the deploy of the workspaces will be done locally in UNPARALLEL on a dedicate machine to provide the needed resources to the users.
- **SmartCLIDE Back-end**: Back-end services will be also deployed locally. It is expected that services such as the ones to index and analyse the services, to be heavily processing and will also run locally in UNPARALLEL. These services will communicate with IoT-Catalogue through the WebSocket for real-time communication (subscription of updates on services) with conjugation with REST API to post information updates.

SmartCLIDE will share with IoT-Catalogue its authentication. In other words, this means that to a user of IoT-Catalogue will be given access to SmartCLIDE IDE features using IoT-Catalogue authentication credentials. And this way, IoT-Catalogue users can seamless access to the integrated development environment provided by SmartCLIDE.

SmartCLIDE takes advantage of WebSocket and a REST API communication to connect to IoT-Catalogue.

### 5.1.4 CONTACT Elements for IoT – CONTACT Software GmbH

With CONTACT Elements for IoT, companies quickly provide solutions that intelligently evaluate the data of industrial assets using a Digital Twin. Elements is a complete platform from edge connectivity to business applications for customers on the web. Elements is ideal for creating agile solutions according to the low-code principle and fulfilling comprehensive requirements for security, cloud and multi-tenancy operation, as well as integration into the company's IT.

#### 5.1.4.1 Continuous Integration

**Why CI in General?**

The effort required to integrate a system increases exponentially with time. By integrating the software system more frequently, integration issues are identified earlier, when they are easier to fix, which is typically right after the actual change that led to the issue has happened and the overall integration effort is reduced. The result is a higher-quality product and more predictable delivery schedules.

Continuous integration provides the following benefits:

- **Improved feedback.** Continuous integration shows constant and demonstrable progress.
- **Improved speed.** A good CI offers high speed: Developers can deploy small increments much faster and get almost instant feedback. This means that other colleagues have to wait much less for the increments, the processing times of a change are shortened and therefore the waiting times are decreased as well. If the waiting times are short, we increase throughput because the queue length is shorter.
- **Improved error detection.** Continuous integration enables you to detect and address errors early, often minutes after they’ve been injected into the product. Effective continuous integration requires automated unit testing with appropriate code coverage.
- **Improved collaboration.** Continuous integration enables team members to work together safely. They know that they can make a change to their code, integrate the system, and determine very quickly whether or not their change conflicts with others.
- **Improved system integration.** By integrating continuously throughout your project, you know that you can actually build the system, thereby mitigating integration surprises at the end of the development project.
- **Improved automation.** Continuous integration forces us to automate all steps necessary for integrating the software, such as building and testing for example (which ultimately paves the path to automatic deployment).
- **Reduced number of parallel changes** that need to be merged and tested.

- **Reduced number of errors** found during system testing. All conflicts are resolved before making new change sets available and by the person who is in the best position to resolve them.
- **Reduced technical risk.** You always have an up-to-date system to test against.
- **Reduced management risk.** By continuously integrating your system, you know exactly how much functionality you have built to date, thereby improving your ability to predict when and if you are actually going to be able to deliver the necessary functionality.
- **Drive with "open eyes".** By providing on each commit the status of the entire code.

Figure 15 shows the development process flow (including the tools in use) at CONTACT Software. All of the applications as well as the platform are developed the same way.

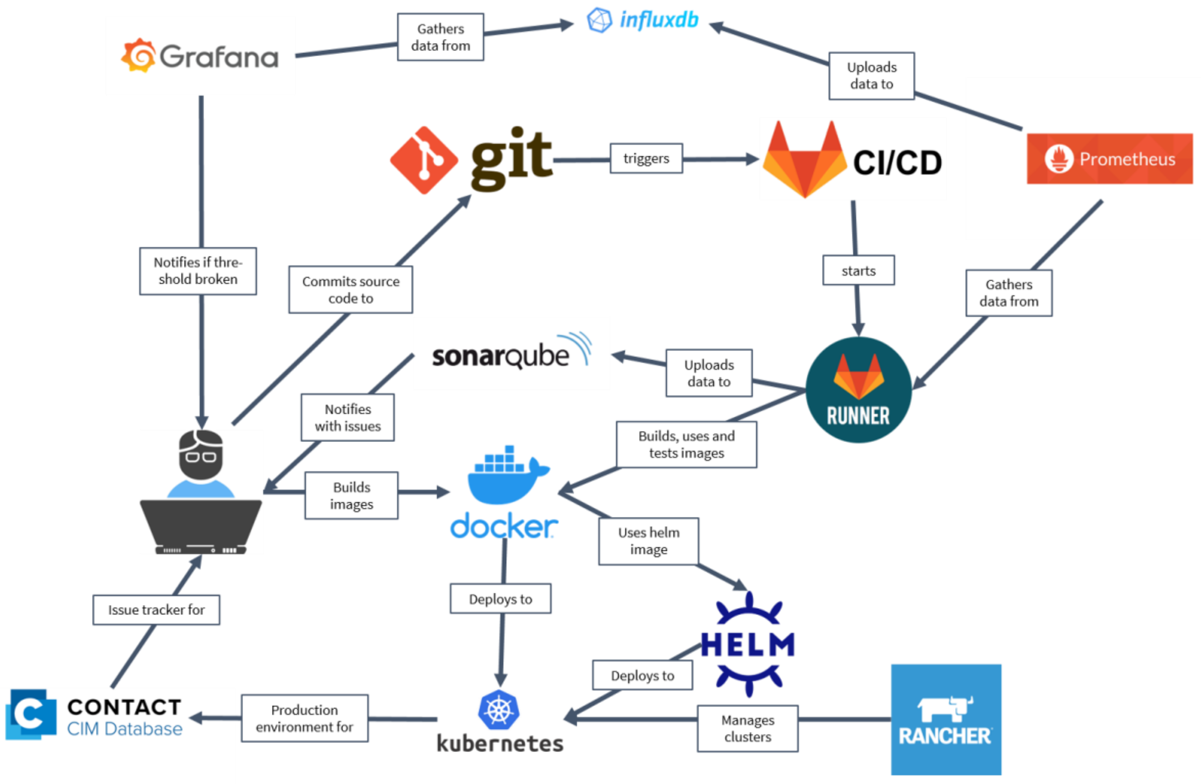


**Figure 15: Development Process Flow**

Each push to the internal GitLab instance triggers a whole CI pipeline (as long as a valid configuration exists). We do this, since we try to follow the principle "If it hurts, do it more often". Since the introduction of CI, the duration of the average pipeline has grown a lot. However, after the migration from Buildbot to GitLab CI, the duration of a typical pipeline was cut more than in half due to the built-in GitLab feature of parallelization. Although the pipelines consume a lot more resources in general doing it that way, it is still cheaper than developers waiting for feedback on their latest commit.

**Pipeline Scope**

The pipeline should contain of the following stages, which will result in a diamond-shaped form.

- **Building** (if necessary): On all supported platforms (typically Windows and Linux and the lowest supported version of CONTACT Elements). The building stage compiles the code (with the newest changes, so the latest revision of your repository) and may contain initial unit testing to ensure that the system works on a technical level. It also builds and signs possible installers (in our case mostly .msi and signing is done with signtool) or respective eggs/ wheels.
- **Testing**: The testing is done automatically and mainly asserts that the system works at the functional and non-functional level, meaning that the behaviour of the system meets the needs of its users. This is where the parallelizing develops its full shape. It is done on all relevant platforms (applications usually on Linux and Windows, CAD packages only on Windows) and include unit, acceptance and performance testing, while collecting coverage and test results. Also some manual testing might be performed, but this is best done after deploying the changes into a test system, to which also stakeholders have access to (which is also part of the automated pipeline).
- **Analysis**: Consists of static code analysis (SonarQube), checking of the CONTACT Elements customizing and security checks (using tools such as Bandit). With SonarQube as our tool of choice and the possibility of creating quality gates, we committed ourselves to a few crucial metrics we set a threshold for that may not be broken by a change. These metrics are firstly critical issues (none allowed), unit test success (100%) and coverage on new code (above 60%). This applies to all branches under development (excluding maintenance branches where only bug fixes are committed). With GitLab CI, we are able to analyze the source code early in the feature branches, so a developer will not be surprised by breaking quality gates after merging changes into the master.
- **Release:** The purpose of the release stage is to deliver changes to the actual users (who may just be a stakeholder in a development project). The artifact in this case does not always have to be a packaged piece of software, but may also be a deploy into a staging environment. Hence, this stage must not always lead to an actual tag, however merges and commits on the master have the upload of a dev release to the package server (PyPi) as a result. The deployment takes place in the inhouse Kubernetes cluster in which each development team has its own namespace to avoid conflicting services to be deployed.

#### 5.1.4.2 Continuous Delivery/ Deployment

While we do not fulfil each criteria when it comes to Continuous Delivery or Deployment (it is not automatically shipped to the customer), we do it in a smaller scope. Usually, each run of a GitLab CI pipeline on a master branch of a repository has an upload of a dev release to the inhouse package repository as a result. On top of that, a docker image with all the current changes is built and uploaded to the internal docker registry (which may be used by Sales for demonstration purposes) as well. Furthermore, each pipeline (also the ones not running on a master branch) automatically creates a test deployment on Kubernetes, which may then be used for smoke tests or a review in a merge request (UI or configuration changes mainly).

However, there are currently two production systems in use to which we deploy continuously. First, the web demonstrator, to which a potential customer may gain access by a simple registration on our website and secondly out inhouse version of CONTACT Elements (used for document, issue and project management for example). These systems are automatically updated by each commit on its master branch.

# 6 System Delivery Plan

## 6.1 Software Engineering Approach

### 6.1.1 Principles and Validation of the SmartCLIDE solution

The proposed software engineering approach will apply a blend of agile/lean frameworks that focus on putting the **user in the centre** of the solution construction process. In this way, by involving end-users from early stages, uncertainty will be reduced. In this sense, partners piloting the solution, and some additional selected customers, will act as Product Owners of the agile development process, thus participating during:

- The gathering of requirements
- The prioritization of requirements to be implemented, based on market value
- The integration, and the validation of the concept and its correct operation.
- Feedback provisioning (back into design and implementation).
- The assessment of the solution from the technical and business perspective.

Following this approach, the design, the implementation and the validation happen in small iterations, i.e. they follow an iterative evolutionary approach. In this way, at the end of each iteration, there will be a working prototype, so that the effectiveness of the solution (methodology and services) will be evaluated by the pilot users.

During the development of the project, SmartCLIDE will provide **business metrics** to showcase the improvement achieved in relation to the software creation process for pilot partners, and other social and business benefits for industrial partners, ICT providers, end users, etc.  Some other interesting metrics would be those related to S/T objectives, so that we can validate some of the expected aspects of the solution: flexibility, extendibility, scalability, effectiveness, etc. The **Early prototype testing** will use the key elements of the envisioned project results as add-ons to the existing processes and infrastructures provided by the pilot users, and will aim at TRL-5. To measure the success of the project technical results, the **full prototype** will be made accessible to the pilot users. Besides testing the full prototype in an experimental environment, many business metrics will be assessed, for example: time to implement a full microservice, time to deploy a microservice, mean team between development and production deployment, number of detected errors by time, learnability or time taken to implement the first application, etc. Finally, the prototype will be configured for the assessment in real-life scenarios. The measurement of all defined metrics and benefits will continue. During this phase, it is expected to achieve all defined targets and a TRL 6.

### 6.1.2 Methodology: Adaptive Project Management approach

For the development of SmartCLIDE, the consortium proposes to follow an Adaptive Project Management (APM) approach. The reason for this is that SmartCLIDE encloses a **high complexity** (due to the combination of leading-edge technologies), as well as **high levels of uncertainty** (due to the application of early software engineering theories that disrupt the current Software Engineering practice) and the APM approach which has been designed to successfully manage complex projects with high levels of uncertainty.

In summary, APM combines the best practices of waterfall and agile. The first one were designed to deal with complexity, but not uncertainty, while the latter were designed to deal with uncertainty but not complexity. In this sense, APM manages risks and tracks the critical path while recognizing that our knowledge is incomplete at the beginning of the project, and it grows as the project advances.

Making use of this methodology, the engineering tasks are organised as follows:

### 6.1.3 Waterfall approach

This approach will be followed during the first stage of the project, when the System Concept is to be designed. During this stage, the consortium will carry out the following tasks:

- Analyse the current state-of-the-art technology and market trends
- Gather requirements from potential end-users.
- Design a generalized concept of the solutions based on the previous tasks.
- Describe the different sub-systems and services involved.
- Describe the features to implement
- Identify and categorize the potential risks associated with hindering the successful implementation of the designed solution.

In the end, this concept will be used as a Minimum Viable Product, that can be validated with the potential end-users during the next stages of the project.

Besides this first stage, when the project comes to an end, the work plan once again follows the waterfall paradigm.

### 6.1.4 Agile approach

Following a SCRUM-like process, the next stages will be fully iterative for both the research activities and the implementation of the technological development. The work to be carried out under these WPs will be organized in monthly Sprints. The scope of each Sprint will be planned before its start, prioritizing the most critical issues (i.e. the issues that reduce the risks of highest impact, and provide the highest value to the end-users). Depending on the purpose, a Sprints can be devoted to:

- **Experimentation.** In this scenario, there can be several experimental/exploratory lanes running in parallel with the purpose of validating different hypotheses, exploring the technical capacities of a technology, or evaluating the viability of a specific feature. At the end of the sprint, the **outcome** would be an **accepted or rejected hypothesis.** If accepted, it could result in further development of a feature in next sprints, or in the adoption of a new technology.

- **Implementation.** When the purpose of the Sprint is non-experimental, the team will schedule the implementation of features at the planning session. The result of these Sprints will be **functional prototypes** of the sub-systems, services or features that will incrementally grow iteration after iteration. The RTD partners responsible for a technical feature will have an end-to-end responsibility, developing the full stack of the feature and testing it to demonstrate its correct operation. The partners acting as end-users of the solution will validate/accept the outcomes of each Sprint in their own contexts.
- Through these iterations we will incrementally gain knowledge about the final solution, and the process itself, providing lessons learned at the beginning of each sprint, and leaving room to schedule the implementation of new requirements (introduced by end-users, the market trends or other research projects).

The prototypes resulting from the research and technological development stages will be integrated into a Full Prototype. This prototype will be validated in real business conditions in order to provide feedback to fix bugs (if necessary). In order to prevent critical issues at the integration stage, **the end-users will be involved in the validation phase of each sprint.**

It is important to remark that some tasks and work packages (such as the Architecture Definition, or WP2 and WP3), are expected to receive intense feedback during the last months of the project, i.e. after they finalize. However, they will remain active until the end of the project, since they can obtain valuable feedback at any time during the project.

## 6.2 Implementation Schedule

The three main releases for the SmartCLIDE platform are:

- Early Prototype in M20
- Full Prototype in M30
- Final Prototype in M36

The implementation schedule envisages that the Early Prototype will contain implementations of each SmartCLIDE component, whereby each component will implement a defined subset of all features that are planned for each component.

The goal for the Early Prototype is to provide a functional prototype, which supports a defined usage scenario that can be tested and evaluated in each of the pilot cases.

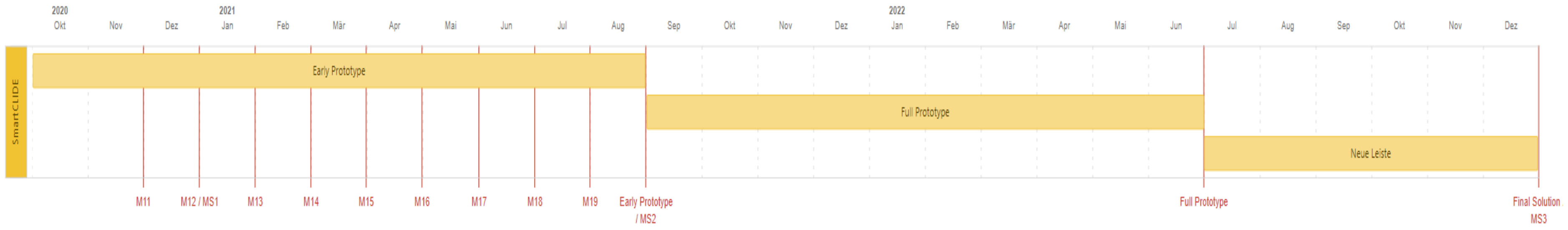


**Figure 16 SmartCLIDE Development Roadmap**

The SmartCLIDE consortium agreed on the following roadmap to reach this goal - see Figure 16 and Table 13 below. The roadmap schedules a milestone for each month on the way to the Early Prototype in M20. Each milestone from M14 on shall result in an integrated intermediate release of the SmartCLIDE platform, which can be deployed and tested by the pilot case partners. Each integrated intermediate release of the SmartCLIDE platform will implement additional functionality as well as take into account feedback from the pilot case partners' testing of the previous release.

**Table 13: SmartCLIDE Development Roadmap**

| Milestone | Description |
|---|---|
| M11 | • common development approach defined and agreed<br>• common source code repository set up<br>• decisions on which existing SW SmartCLIDE components may be based on finalised |
| M12 | • usage scenario defined<br>• features to develop for EP selected<br>• SmartCLIDE runtime environment set up<br>• development infrastructure set up |
| M13 | • specification of internal cross-component APIs for EP finalised<br>• specification of public API for EP finalised |
| M14 | • first version of backend implemented, containing only the internal cross component API and public API endpoints for EP, with mocked backend functionality<br>• first version of user interfaces implemented, displaying mocked data<br>• integrated SmartCLIDE platform version EP-alpha-1 |
| M15 | • APIs for EP refined and finalised<br>• second version of backend functionality implemented<br>• second version of user interfaces implemented<br>• integrated SmartCLIDE platform version EP-alpha-2 |
| M16 | • third version of backend functionality implemented<br>• third version of user interfaces implemented<br>• integrated SmartCLIDE platform version EP-beta-1 |
| M17 | • fourth version of backend functionality implemented<br>• fourth version of user interfaces implemented<br>• integrated SmartCLIDE platform version EP-beta-2 |
| M18 | • fully integrated SmartCLIDE platform version EP-RC-1, containing all features planned for EP (feature-complete) |
| M19 | • test and evaluation of SmartCLIDE platform version EP-RC-1 by pilot case partners<br>• fully integrated SmartCLIDE platform version EP-RC-2 |
| M20 | • fully integrated SmartCLIDE platform version EP-RELEASE |

The SmartCLIDE Full Prototype will be based on the Early Prototype, and implement the remaining functionality, which has not been included in the Early Prototype. It will in general follow a similar roadmap as defined for the Early Prototype, but make necessary adjustments based on experiences from Early Prototype development. A detailed roadmap will be published on the SmartCLIDE repository / wiki after finalising the Early Prototype (M20).

For the Final Prototype in M30, the consortium will focus on implementing feedback from the Full Prototype assessment and continue to release intermediate versions of the SmartCLIDE platform on a monthly basis. A detailed roadmap will be published on the SmartCLIDE repository / wiki after finalising the Full Prototype (M30).

# 7 Conclusions

The present deliverable followed the conceptualization and system work conducted in Tasks 1.2, 1.3 and 1.4, and capitalized on their outcomes, that is D1.2, D1.3 and primarily D1.4, and proceeded to finalize this work by presenting in detail the conceptual/logical components-based architectural view, the information flows/communication view, the system deployment view, providing details on the current practices in terms of deployment at pilots, as well as the delivery plan/development roadmap of the SmartCLIDE platform. The requirements engineering, design and architecture specification, within agile development environments and as foreseen in SmartCLIDE workplan, is an ongoing process, thus it is expected that updates will be introduced within the duration of the project.

## References

1. OAuth 2.0 (2020). Available at: https://oauth.net/2/ , visited on 2020-09-10.

2. 2019. Flask, 2019 (2019). Available at: https://palletsprojects.com/p/flask/ , visited on 2019-10-18.

3. 2019. Python Eve, 2019 (2019). Available at: http://docs.python-eve.org/en/stable/ , visited on 2019-10-16.

4. MySQL (2020). Available at: https://www.mysql.com/ , visited on 2020-10-09.

5. 2019. MongoDB, 2019 (2019). Available at: https://www.mongodb.com/ , visited on 2019-10-10.

6. Apache Kafka (2019). Available at: https://kafka.apache.org/ , visited on 2019-12-12.

7. Apache Qpid (2020). Available at: https://qpid.apache.org/ , visited on 2020-10-09.