EUROPEAN LANGUAGE GRID

D1.1

Requirements and Architectural Specification of the Base Infrastructure

Authors:	Florian Kintzel (DFKI), Maria Moritz (DFKI), Georg Rehm (DFKI)
Dissemination Level:	Public
Date:	30-04-2019

About this document

Project	ELG – European Language Grid
Grant agreement no.	825627 – Horizon 2020, ICT 2018-2020 – Innovation Action
Coordinator	Dr. Georg Rehm (DFKI)
Start date, duration	01-01-2019, 36 months
Deliverable number	D1.1
Deliverable title	D1.1 Requirements and Architectural Specification of the Base Infrastructure
Type	Report
Number of pages	21
Status and version	Final – Version 1.0
Dissemination level	Public
Date of delivery	Contractual: 30-04-2019 – Actual: 30-04-2019
WP number and title	WP1: Grid Platform – Base Infrastructure
Task number and title	Task 1.1: Requirements collection, architecture specification, base software selection
Authors	Florian Kintzel (DFKI), Maria Moritz (DFKI), Georg Rehm (DFKI)
Reviewers	Ulrich Germann (UEDIN), Jana Hamrlova (CUNI)
Consortium	Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Germany
	Institute for Language and Speech Processing (ILSP), Greece
	University of Sheffield (USFD), United Kingdom
	Charles University (CUNI), Czech Republic
	Evaluations and Language Resources Distribution Agency (ELDA), France
	Tilde SIA (TILDE), Latvia
	Sail Labs Technology GmbH (SAIL), Austria
	Expert System Iberia SL (EXPSYS), Spain
	University of Edinburgh (UEDIN), United Kingdom
EC project officers	Philippe Gelin, Alexandru Ceausu
For copies of reports and other ELG-related information, please contact:	DFKI GmbH European Language Grid (ELG) Alt-Moabit 91c D-10559 Berlin Germany Dr. Georg Rehm, DFKI GmbH georg.rehm@dfki.de Phone: +49 (0)30 23895-1833 Fax: +49 (0)30 23895-1810 http://european-language-grid.eu © 2019 ELG Consortium

Table of Contents

List of Figures

List of Terms

Cluster (Kubernetes)	A set of machines, called nodes, that run containerized applications managed by Kubernetes.
Container (Kubernetes)	An instance of an image.
Controller (Kubernetes)	A control loop that watches the shared state of the cluster through the API server and makes changes attempting to move the current state towards the desired state.
Deployment (Kubernetes)	A Kubernetes configuration that manages multiple replicas of an application. If one instance fails or becomes unresponsive, it is automatically replaced by the Deployment.
Deployment (general)	The action of installing a software package on a given server.
Docker	A software technology providing operating-system-level virtualization also known as containers
Image	A bundle of one or more software components (data, executables) that provide a service through clearly defined access points (ports) on the host while running in isolation from the surrounding environment otherwise.
Infrastructure-as-code	Infrastructure as code (IaC) is the process of managing and provisioning computer data centres through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.
Instance (Cloud Provider)	A typical cost calculation unit as delivered by cloud IaaS providers. An instance can come in different sizes, e.g., small, medium, large with respect to cores and RAM available on it. The instance is represented by a virtual machine with the respective hardware resources.
Kubelet (Kubernetes)	An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
Namespace (Kubernetes)	An abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster.
Node (Kubernetes)	A worker machine in Kubernetes.
Object Storage	A data storage architectures that manages data as objects with unique identifiers.

Pod (Kubernetes)	Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. They comprise one or more containers that share storage and network communications, which are otherwise isolated from the pod's host.[1]
S3	An object storage solution developed by Amazon. Today, often used to refer to solutions by other providers that have APIs compatible to the Amazon S3 API.
Service (Kubernetes)	An API object that describes how to access applications, such as a set of Pods, and can describe ports and load-balancers.
Volume (Kubernetes)	A directory containing data, accessible to the containers within a pod.

List of Acronyms

API	Application Programming Interface
CI	Continuous Integration; the process of continuously testing software as it is being developed.
GKE	Google Kubernetes Engine
GUI	Graphical User Interface
HPA	Horizontal Pod Autoscaler. A standard Kubernetes controller that can spin up or tear down container based on live metrics indicating the load a service is under (e.g., CPU utilization).
IaaS	Infrastructure as a Service
ITIL	IT Infrastructure Library
LR	Language Resource. Typically, a data set or corpus.
MQ	Message Queue. A component of the process communication in a software system.
MVP	Minimum Viable Product. A product with just enough features to satisfy early customers, and to provide feedback for future product development.
UX	User Experience. The look and feel that a user experiences when she uses the platform.
VM	Virtual Machine

[1] From https://kubernetes.io/docs/concepts/workloads/pods/pod/

Summary

This document describes the infrastructure design of the ELG platform. It first explains the purpose of the Grid Platform and how it relates to the scope of the project (Sections 1 and 2). From there, the report develops infrastructure requirements (Sections 3 and 4) and presents and justifies architectural design decisions for the platform (Sections 5-7). The sheer size and managing effort that comes with an endeavour such as ELG requires multiple stages of decision making. Temporally, these refer to, (i) the proposal writing phase, (ii) the architecture planning phase at the beginning of the project, (iii) later points within the project's runtime. The latter is driven by the agile character of today's software and infrastructure development.

Section 1 motivates the work, gives an entry point on cloud development and show an overview of the ELG's architecture. Section 2 places the architecture within the whole project and details main objectives. Section 3 lists and reasons architectural requirements collected at this early stage of the project. These were mainly collected during discussions with the project partners and experienced researchers working in the field. Section 4 discusses the requirements collected during the selection of a suitable infrastructure provider and introduces the selected provider with whom we are currently setting up a contract, and, consequently, the first grid instance. Section 5 gives an overview of the architecture components on a high-level base. Section 6 contains a detailed outline of the technology for the infrastructure services that represent important elements of the infrastructure architecture. Section 7 introduces the technology stack that we use to build, deploy, and maintain the infrastructure. Finally, Section 8 summarizes the document and outlines the next steps.

1 Introduction – Background – Context

1.1 Cloud Services and Kubernetes

Cloud computing enables on-demand availability of resources, be it CPU power or storage usually offered to a user by a cloud service provider. Main advantages for customers are the lack of need to maintain their own hardware and computing resources. For enterprises it is also a matter of costs and optimized utilization of resources. The ELG is one such endeavour that will, ultimately, provide a wide range of LT services, offered and managed in a scalable manner.

Cloud-based applications currently experience a renaissance in the domain of software architectures and development. The birth of general-purpose cloud services can be dated back to 2006 (Kratzke & Quint, 2017)[2]. While back then, developers focused on managing VMs to host services, the focus shifted to light-weight containerization with the introduction of Docker. More recently, technologies like Docker Swarm, Apache Mesos and Kubernetes significantly simplified the orchestration of large amounts of containers. Kubernetes automates the allocation of hardware to the services running in a grid application, monitors services and traffic, routes and balances compute load, scales services up or down according to their load, and ensures redundancy and uptime of services by automatically replacing units of service (pods) if they fail or become unresponsive.

[2] Kratzke, Nane, and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing – a systematic mapping study." *Journal of Systems and Software* 126 (2017): 1-16.

The base infrastructure is a fundamental part of the ELG. It ensures the availability of the LT services, tools and components provided by the ELG. It is responsible for running the grid and scaling resources to a service's highly heterogeneous properties. This means that it must offer a flexible way of allocating resources to services depending on their hardware needs. For example, services can come with a high need for memory and CPU. But, depending on the task (e.g., training, fitting, or testing) these ratios of memory and CPU can vary substantially. An important task of the base infrastructure is to run a large number of services (up to several hundred in the last year of the project) in a resource-economic manner while still addressing their resource-specific needs.

1.2 Architecture Overview

The overall architecture of the ELG, shown in Figure 1, is structured into four layers:

1. Base Infrastructure (WP1)
2. Platform (WP2)
3. Application Programming Interfaces (API; WP4)
4. Graphical User Interface (GUI; WP3)

This specific diagram visualizes the development roadmap until September 2019; a very first prototype and MVP will be demonstrated at the first annual ELG conference in October 2019. In the figure, development priorities are indicated with labels from A (highest priority) to C (lowest priority). In what follows, we discuss details regarding the architecture design of the base infrastructure. The lower layer contains the nodes that run the Language Technology services, the file and object stores that hold the language resources and data sets. Section 5 will present the architecture in all its facets and details.

For encapsulation purposes of API requests, we foresee to share the API design between the services and the infrastructure. An additional general service API enables the encapsulation of more detailed APIs and their requests. This API is generic in the sense that all services deployed in the grid can be targeted via this API. It is envisioned to be integrated on the platform layer (see Generic API in the Platform Layer in Figure 1).

The architecture diagram was prepared in one of the sessions in the ELG technical kick-off meeting, which took place on 18/19 February 2019 at DFKI Berlin. In addition to the DFKI team, the ELG partners ILSP, USFD and TILDE participated in the meeting, in which the main technical aspects of WP1, WP2, WP3 and WP4 were discussed. The diagram, priorities and next steps are one of the main tangible results of this meeting.

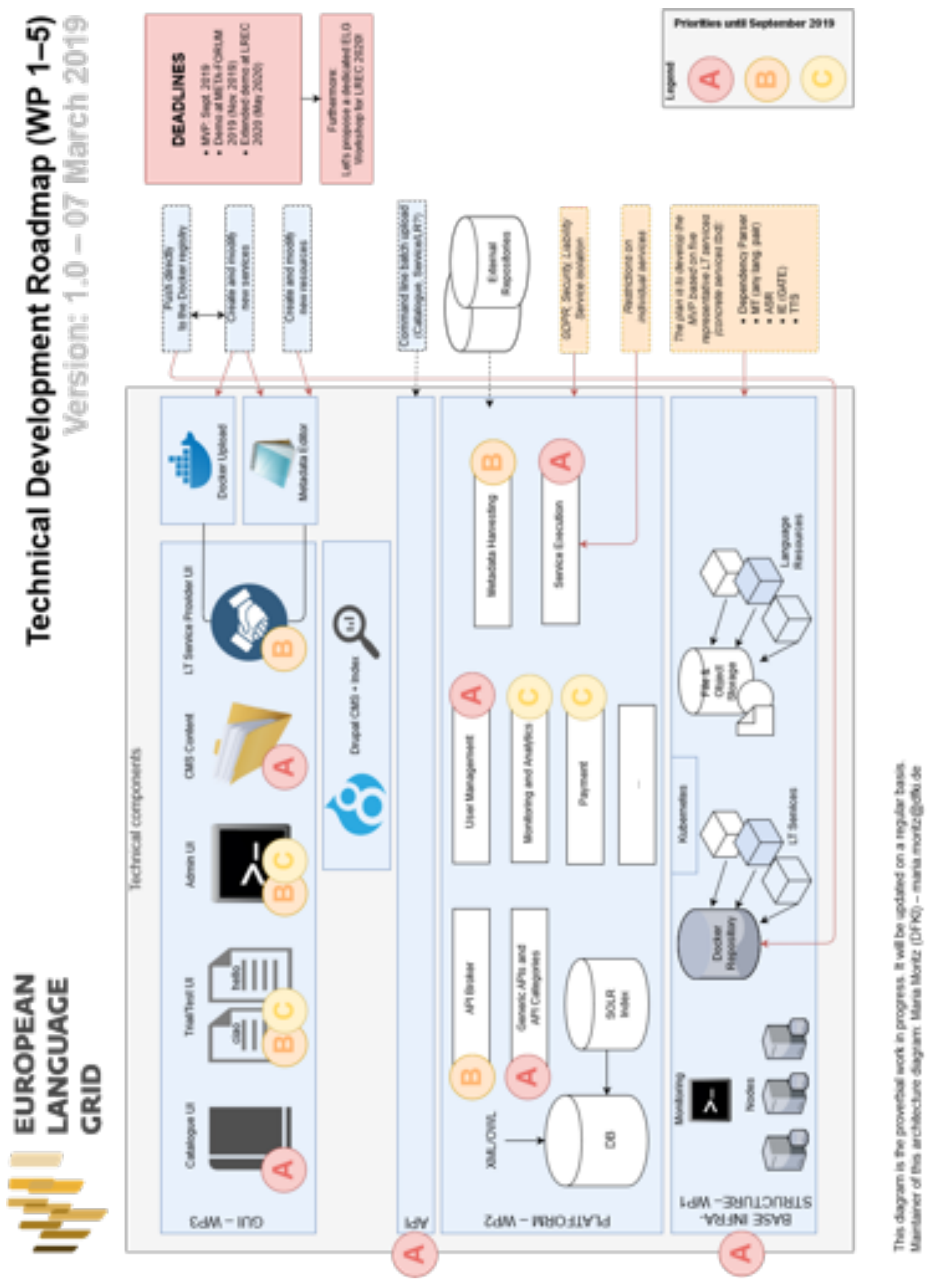


Figure 1: Overview of the architecture of the European Language Grid

1.3 ELG as a Cloud Application

We develop the ELG platform in a cloud-based manner where many LT services run inside Kubernetes pods distributed over a grid of compute nodes, where each pod runs one or more Docker containers. Section 5 describes how the components of the infrastructure interact.

We envision three separate ELG systems for the three different stages, i.e., development, integration and production. We plan for the integration platform to be online in 2020 and the production platform in 2021, respectively. The development cluster will be kept online during the whole project runtime, integration from 2020 onwards and production from 2021 onwards.

2 Platform Infrastructure – Objectives

2.1 Key Objectives

The main objective of the ELG is to provide companies, research organisations, and industry with a reliable and scalable platform through which they can offer, showcase, test, and evaluate their technologies to interested stakeholders, typically from industry and research, but also including public administrations and other organisations. These technologies mainly include Language Technology services, data sets, technologies, tools, and language models created by industry or research organisations. Once established, the ELG also represents an important marketplace for both, including the exchange of established and emerging Language Technology services, and a source to enable its further development, improvement, and maintenance. The European Language Grid will make progress in a number of important areas:

1. The ELG will serve as the primary one-stop shop for European Language Technology industry and research, i.e., a central service and data set catalogue and marketplace that is always up to date.
2. The ELG will constantly yield recent research and development results towards European LT industry.
3. The ELG will offer benefits for reproducible research (especially with regard to Language Technology and Computational Linguistics research), and promote both Open Science and Open Source.

To this end, we will design, develop and deploy the ELG as a unique grid architecture that will encapsulate basic and complex Language Technology services in atomic containers. This approach solves the notorious issue of interoperability and ensures that services, tools and applications remain usable, no matter which (software, library, package, version) dependency they require. The technological approach upon which the ELG platform is based enables innovation and synergies between commercial and non-commercial Language Technology users and providers that – to this extent – have not been possible in the past.

2.2 Distinction from Other WPs

The ELG base infrastructure is the foundational layer of the whole platform. The infrastructure ensures its smooth running by serving as an interface to the dynamically allocated hardware resources. This allocation is critical and key to powering a distributed, heterogeneous cluster of Language Technology services and tools. Hence, the infrastructure not only provides the necessary resources required by a huge amount of software tools, it also ensures the economical usage of these resources by monitoring running nodes and deciding whether new images need to be deployed or “underperforming” images need to be terminated. To meet the requirements (see Section 3), the base infrastructure’s design makes a series of decisions that help to successfully build up the upper layers of the system.

3 Infrastructure Requirements and Compliance Considerations

The following requirements are based on the grant agreement, the architecture roadmap as defined in the technical kick-off meeting, and discussions within the Infrastructure Task Force.

3.1 Requirements for the Base Infrastructure

A cloud environment’s primary advantage is its elasticity. Following Kratzke & Quint (2017), elasticity is understood as the degree to which a system adapts to workload changes by provisioning and deprovisioning resources automatically. Requirements for the base infrastructure affect how software is stored, built, operated and distributed in the ELG:

1. *Dockerizable and isolated* – Services that need to be ingested into the grid are supposed to be dockerizable. For the time being, only Linux containers will be supported.
2. *Distributed* – The architecture of the grid is distributed. Services and software code will run on a number of different nodes depending on both the hardware available in the grid, and the hardware resources that are necessary to run a service.
3. *Scalable* – While the response time (latency) of a particular service depends on its implementation, the platform must provide sufficient scalability to handle increased demand without deterioration in the quality of service a client receives. The response time depends on the technical internals of a service.
4. *Offering an appropriate RAM/CPU ratio for Language Technology services* – This requirement is covered by the selection of the cloud infrastructure provider (see Section 4) who delivers the cluster according to figures for which we, i) estimated the needed core equipment and resources, and ii) can be upgraded dynamically on demand.

3.2 Requirements for the API / Interface Design

APIs handle communication data throughout the whole infrastructure.

5. *API encapsulation* – We foresee an additional, generic service API on the platform layer additionally to the service API exposed by a given LT service. This API is generic in the sense that all services deployed in and by the grid can be targeted via this API. The service itself only needs to do minimal adaptation. The infrastructure simply pipes service requests through a request dispatcher towards the MQ.
6. *Service Adapter* – The service adapter is a small module added to each service that implements the generic API. Its purpose is to pass requests smoothly to and from the more detailed service APIs. It can run on a separate pod (also known as a sidecar), in a separate container in the same pod.
7. *Service API publishing* – The majority of the APIs exposed by LT services are expected to not be published directly to the internet. This is mainly necessary to ensure security within the grid, and to enable the injection of additional processing logic into the request processing (e.g., validation and authentication). Further, the respective container that handles a given API request might not be initialized yet, so storing the request temporally will be needed (see Section 5.2).

4 Provider Selection

4.1 Budget Distribution

From Cloud Spectator's 2018 report on *Top 10 Public Cloud IaaS Providers*[3] and its report on *2017 Top 10 European Cloud Providers*[4], we first collected rough figures on instance[5] pricing offered by well-known international and national cloud service providers including Google, Azure, Amazon and 1&1. Based on these figures and considering the budget of the subcontract foreseen in the grant agreement (100,000.00€), we first estimated

[3] https://connect.cloudspectator.com/2018-top-10-cloud-iaas-providers-benchmark-pdf-download

[4] https://connect.cloudspectator.com/2017-top-european-cloud-providers-report

[5] Common instance sizes that providers usually offer are Small, Medium, Large and Extra Large instances. We choose Large as the instance size we base our calculations on.

an instance prize of 2886 Euro a year (for a standard large instance size, eight cores with 16GB RAM). We assume a budget split for the project runtime of about 1/9 to 3/9 to 5/9 over the three years (see Table 1).

	Split	Budget split	Instances	Cores	RAM (GB)	Storage (GB)
Year 1	1/9	11000.00€	4	32	64	800
Year 2	3/9	33000.00€	12	96	192	2400
Year 3	5/9	56000.00€	20	160	320	4000

Table 1: Budget split for the project runtime (implicit ramp-up phase of the ELG platform)

These figures serve as a first indication to enable us to select a provider, negotiate prices and split the resources among the three clusters (i.e., development, testing, production) that we will work with during the project. In the final contract the 1/9 to 3/9 to 5/9 ratio is still considered.

4.2 Provider Comparison

The market for Managed Kubernetes solutions in Europe is still young, most providers have launched their offers only in 2018. When we started our research it quickly became clear, that there are big differences between the different "Managed Kubernetes" offerings that can only be identified when looking into the respective specifications in detail and experimenting with the platforms. Even the more mature IaaS market is showing differences in, e.g., price/performance ratios of up to 400% (see, e.g., the reports from cloudspectator.com). While the bare core/RAM numbers are easy to compare, providers can easily "tweak" their offerings by utilizing core over-provisioning, i.e., selling the same core multiple times. Managed Kubernetes adds an additional layer on top. It takes over the setup, maintenance work and availability of infrastructure services in the form of a platform offered as part of a managed solution. The following providers were included in the comparison:

- Deutsche Telekom AG
- OVH SAS
- Ionos SE
- SysEleven GmbH
- ClaraNet Ltd.
- Netways GmbH
- Servinga GmbH
- ScaleUp Tech GmbH
- Unbelievable Machine GmbH
- CloudSky GmbH
- Noris Networks AG

These providers were compared regarding the following criteria.

4.2.1 European Provider

While US providers have the technical lead in Managed Kubernetes, we perceive the ELG, as a quintessentially European project, to operate on a European system, provided by a European company. We excluded all providers located outside of Europe, as well as all providers that internally make use of providers not located in Europe. A regional provider comes with the additional advantage of low communication overhead and low traveling costs, for example, when travel to trainings and workshops is necessary.

4.2.2 Managed Kubernetes

For each of the shortlisted providers, a test installation was requested for a hands-on look at the platform. Most (but not all) providers agreed to this free of charge. Providers that did not provide the test installation in time, or not free of charge were removed from the shortlist.

The test installations were evaluated by installing a small set of services and examining the tooling that is available out of the box. The providers were then compared qualitatively with the de-facto industry standard Google Kubernetes Engine (GKE) in terms of usability and features. While none of the examined provider offerings do reach the GKE standard to 100%, there were huge differences in the amount of service offered.

The main technical features that were identified to be mandatory are:

- object storage compatible with the standard of the Amazon S3 REST API[6]
- flexible instance sizes, ratios of 1:4 cores/RAM or higher preferred
- possibility for clustering and geo-redundancy (for later phases in the project)
- HDD as well as SSD volumes
- Kubernetes maintenance support (e.g., updates)

4.2.3 Consulting

The likely need for consulting services for the infrastructure setup had already been identified in the proposal. We focused on providers who also offer consulting services alongside Managed Kubernetes.

4.2.4 Experience

Most European providers are relatively new in the market. Therefore, Managed Kubernetes is not yet a mature market in Europe. We excluded some providers deemed too small or not experienced enough with Kubernetes (some were in public beta stage with their offerings). These were Servinga, Ionos SE as well as Netways.

4.3 Shortlisted Providers

Table 2 shows the providers that we asked to send us an offer for an averaged cluster instance. All of them build upon the OpenStack platform and offer Managed Kubernetes as resource control management. Table 2 orders the shortlisted providers in descending order of preference. Geographical redundancy, customizable instance sizes, and S3 as object storage are the most important requirements. Clustered computing centres and a free consulting service constitute the next layer of parameters for our decision making. Certainly, a German provider that is close to the Berlin site of DFKI is preferable in order to minimize travel expenses in case face-to-face meetings or training sessions are necessary. Finally, an important discriminating feature is the price. The offers we initially received were highly diverse regarding service detail level, instance size, and, hence, pricing. Thus, to make comparison clearer and easier for us, we requested an offer for a setup, including 30 cores, 120GB RAM, 1TB object as well as block storage, for which we detailed the resource equipment upfront. The pricing sent to us is shown in Table 2, Column 2.

---

[6] Details are available at https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

	Offer (per month, approx.)	Locations	Clustering	Geo redundancy	Consulting	Live since	Instance size	Storage	Object Storage API
SysEleven	**1720€**	Berlin	yes	planned 2019	yes	2018	1:4, custom	QuoByte	S3
Netways	**1100€**	Nürnberg	–	planned 2019	extra costs	2018	custom	Ceph	Swift, S3
Servinga	**1780€**	Frankfurt	roadmap 2019	planned 2019	yes	2018	fixed, custom	Ceph	S3
ScaleUp Tech	**1240€**	Hamburg, Berlin, Düsseldorf	–	yes	yes	n.a.	1:2	Ceph	Swift
OVH	–	Paris; Kubernetes France only	–	planned	–	2019	custom	–	S3
Unbelievable Machine	**3700€**	Berlin	–	–	yes	n.a.	1:4-1:8	–	–

Table 2: Shortlisted providers

The sizing of this setup was agreed upon to be sufficient for the initial development stage, i.e., the first 12 months of the project. Towards the end of this time frame, we will re-evaluate both the hardware requirements of the grid, as well as the support, consulting and operational performance of the selected provider. As the grid infrastructure will be configured within a central repository (see Section 7.1.2), it will be easy to move the development cluster to another provider if the current one is found lacking in performance or support. Also, as the market for Managed Kubernetes is rapidly evolving in Europe, it may well be that new offers with superior price-performance will emerge. In any of those cases, we will reconsider the provider selection. Technically, this will be supported through the use an of infrastructure-as-code approach to cluster configuration.

4.4 Provider Selection

The main driver for the selection was the evaluation of the test installation. A few providers were removed from further consideration at this stage, because they did not give access to a test installation free of charge or in time (UM, OVH, ScaleUp, Telekom, Noris Networks).

Among the others, we observed vast differences in the maturity of their Kubernetes setups. The most sophisticated solutions were provided by the companies SysEleven GmbH and Servinga. Ionos SE and Netways were removed from the list afterwards. We also removed the providers CloudSky and ClaraNet, that, while based in Europe, utilize data centres owned by US-based companies and only re-sell those with a managed Kubernetes solution on top. Offerings of the remaining companies (Servinga GmbH, SysEleven GmbH) were then compared based on price, consulting capabilities, and experience in hosting Managed Kubernetes.

Eventually, we selected SysEleven GmbH as the provider for the first development phase of the platform (testing and production will come at a later stage). SysEleven offers a mature cloud platform, a competitive price and demonstrates above-average technical expertise in Kubernetes know-how. This is also evidenced by SysEleven being a member of the Cloud Native Foundation, being BIS- and ISO certificated, and being the only provider on the shortlist that is mentioned as a hosted solution on the Google Kubernetes website directly. They offer all mandatory platform features and are one of the few providers offering Kubernetes clustering support (geo-redundancy with a federated Kubernetes cluster). Additionally, SysEleven hosted a hands-on workshop free of charge for DFKI, showcasing their solution. Their consulting services may be utilised later.

We are currently finalizing the contract with SysEleven. For the development cluster, we ordered 15 cores/60GB RAM per month while additionally being able to book the same amount on top on demand.

5 The Architecture in Detail

5.1 On Demand Service Instantiation

As the platform is expected to host hundreds of services eventually, it was deemed unfeasible to keep all services instantiated all the time (even with just a single instance). Even a conservative estimate of 1 core / 2 GB RAM for each service instance would lead to hundreds of cores and enormous amounts of RAM necessary for an idle system. The alternative is to instantiate the services on demand and only if requests for them have arrived. This approach has a drawback, however, i.e., the spin up time required to start a container causes the execution of the initial request to be delayed until it can be processed.

As a remedy, a configuration is envisioned where containers are instantiated on demand, while they are kept running for a configurable amount of time, even when no requests are pending. The spin up time, thus, will only be an issue when the service is woken up. Still, for some services (e.g., commercial services with fixed SLAs) the drawback of higher resource consumption might be necessary to stick to the SLAs.

Therefore, three options of service instantiation are considered:

1. *Always on:* Container is always running with at least one instance. Highest resource consumption, best responsiveness.
2. *On demand:* Container is spun up on demand only and terminates once the current request has been processed.
3. *On demand with grace period:* Container is spun up on demand and kept running for a configurable amount of time, even when no new requests are pending.

The requests workflow for enabling this is shown in Figure 2.

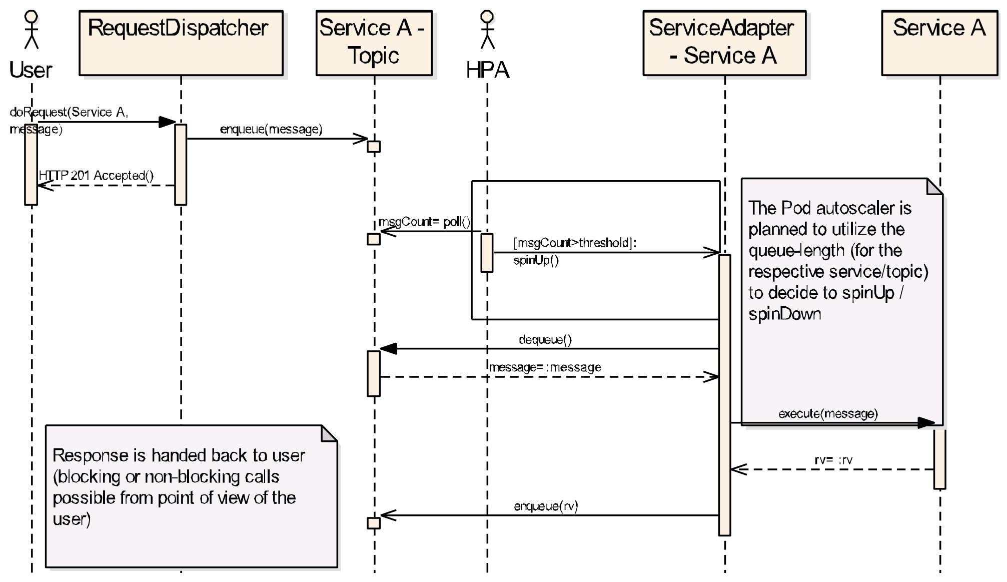

User
RequestDispatcher
Service A - Topic
HPA
ServiceAdapter - Service A
Service A
doRequest(Service A, message)
enqueue(message)
HTTP 201 Accepted()
msgCount= poll()
[msgCount>threshold]: spinUp()
The Pod autoscaler is planned to utilize the queue-length (for the respective service/topic) to decide to spinUp / spinDown
dequeue()
message= :message
execute(message)
rv= :rv
enqueue(rv)
Response is handed back to user (blocking or non-blocking calls possible from point of view of the user)

Figure 2: Overview of the request workflow

The decision to allow containers to scale down to zero replicas directly leads to the need to store requests until the respective container has been instantiated. For this reason, it was decided to include a Message Queue service (MQ) in the architecture. This blends well with LT services offering asynchronous APIs.

For the implementation of the on-demand instantiation, we plan to use the Kubernetes Horizontal Pod Autoscaler (HPA) to spin up Pods. The HPA is not (yet) natively able to scale down Pods to zero replicas, there are, however, extension projects (see, e.g., Autoscaling with Kubernetes[7]), that show the possibility to adapt the HPA to scale down to zero, and to use a custom metric – the MQ length – to trigger up-/downscaling.

5.2 Service API

The direct approach for exposing Language Technology service APIs to users is via Kubernetes' Ingress Controller. This is an API gateway which routes requests to one of the instances of a specific service that runs inside the grid. This is the mechanism most commonly used for REST based APIs (see Figure 3).

However, the notion of having requests queued and executed asynchronously inside the ELG system as explained in the previous section, warrants a different approach to API exposure. The actual API or technical interface that the service exposes to the ELG system is not necessarily the same as the one exposed to the outside world. A service can expose its (public) API, in case it offers one, be it REST or otherwise, using a separate container, hosting only the API, but no execution logic. Its task is to accept and validate the request and, finally, to hand it over to the respective MQ topic. From there, it is handed further, on to the processing service, once an instance of this service comes online.

Connectivity from the MQ to a specific service is provided by a new component, the service adapter, which is – with regard to the current design – envisioned as being the same for all types of services. It forces service developers to adhere to a yet to be designed platform API to be usable from the grid. Figure 3 shows an additional API, i.e., the generic service API (top left corner).

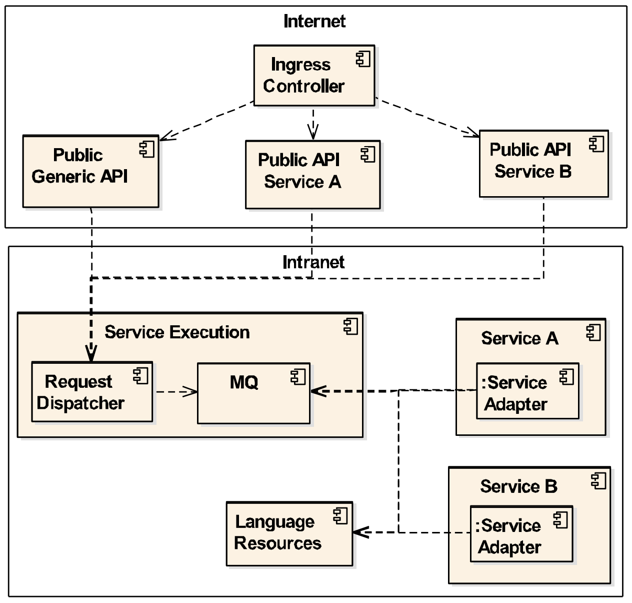


Figure 3: Architecture overview focusing on the API distribution

[7] https://medium.com/@dmaas/producer-consumer-queue-with-autoscaling-on-google-kubernetes-engine-d5859b2f596c

All services deployed in the grid can be targeted via this API. However, by necessity, the API will not offer the same user experience as one explicitly designed for a specific service. It is intended to be a fallback for services not (yet) exposed via a specific API as well as for testing purposes.

5.3 Deployment and Language Resources

Figure 4 shows a visualisation of the deployment of the European Language Grid. The diagram is an illustrative example. In reality, the node choice will be taken over by Kubernetes.

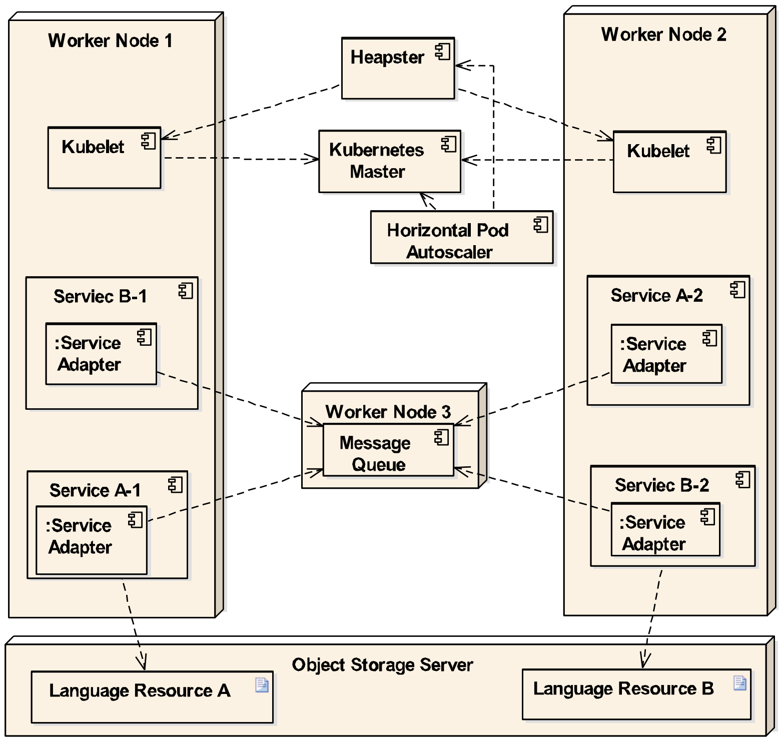


Figure 4: Overview of base infrastructure deployment

Not yet included in the figure is how LRs will be stored. The following solutions have been discussed:

1. *Storing LRs in S3-compatible object storage*: This approach enables the LRs to be uploaded and accessed via HTTP. However, they would need to be copied from the object storage to the working directory of a container to be used during the processing; it must be noted that the idea of making LRs stored in the ELG available to services available in the ELG is not one of our key priorities.
2. *Storing LRs as part of the container image*: LRs can be accessible directly by the service, but this would lead to much larger images and to a highly increased spin up time for containers. Also, multiple instances will require their own instance of the model, which is wasteful. This approach is deemed feasible for small resources only.
3. *Hiding LRs behind an accessor service*: Access to data from the resource is encapsulated behind that accessor service. Access to resource data requires using the service API. This allows scaling the service that uses the model, and scaling the model itself independently from each other.

For the first iteration of the grid infrastructure, the S3 object storage approach was chosen, as it natively allows for resource uploads. In later iterations, when the interaction between LT services and LRs is defined in more detail, it is likely that the two other approaches will also be explored and used in parallel depending, e.g., on the size of the LR data, and on the amount of data needed to handle a service request. Alternatively, the service could implement a layered approach, i.e., keep frequently used data locally, request less frequently used data on demand.

6 Infrastructure Services

The ELG base infrastructure will offer a number of core services, described below, that facilitate debugging, monitoring, and maintenance of the infrastructure as well as individual services. LT service developers will need to follow certain implementational conventions, e.g. , writing logs to *stdout* for Kubernetes basic logging.

This section provides an overview of services that we foresee to be part of the infrastructure layer of the ELG. Thus, they will be automatically available to all Language Technology services to be deployed, forming the backbone of the ELG grid infrastructure.

6.1 Logging

In the first release of the ELG infrastructure, logging will be limited to Kubernetes' default logging facilities (Kubernetes Basic Logging). With this approach, containers log to *stdout* where the output is collected by Kubernetes. These logs can be accessed via the command *kubectl logs* while the user is connected to the cluster. The logs can only be accessed until the pod holding the container is evicted (terminated and removed from the node), so, they are only accessible in a reliable way while the container is running. For a later iteration of the logging services, we will configure a persistent solution with a logging agent, e.g., the Elastic Stack.

6.2 Kubernetes Dashboard

The Dashboard (see Figure 5) is part of every Kubernetes installation. It is a web GUI in which authorized users can inform themselves about details of the Kubernetes cluster. The dashboard provides a subset of the functionalities of *kubectl* CLI in a more user-friendly way.

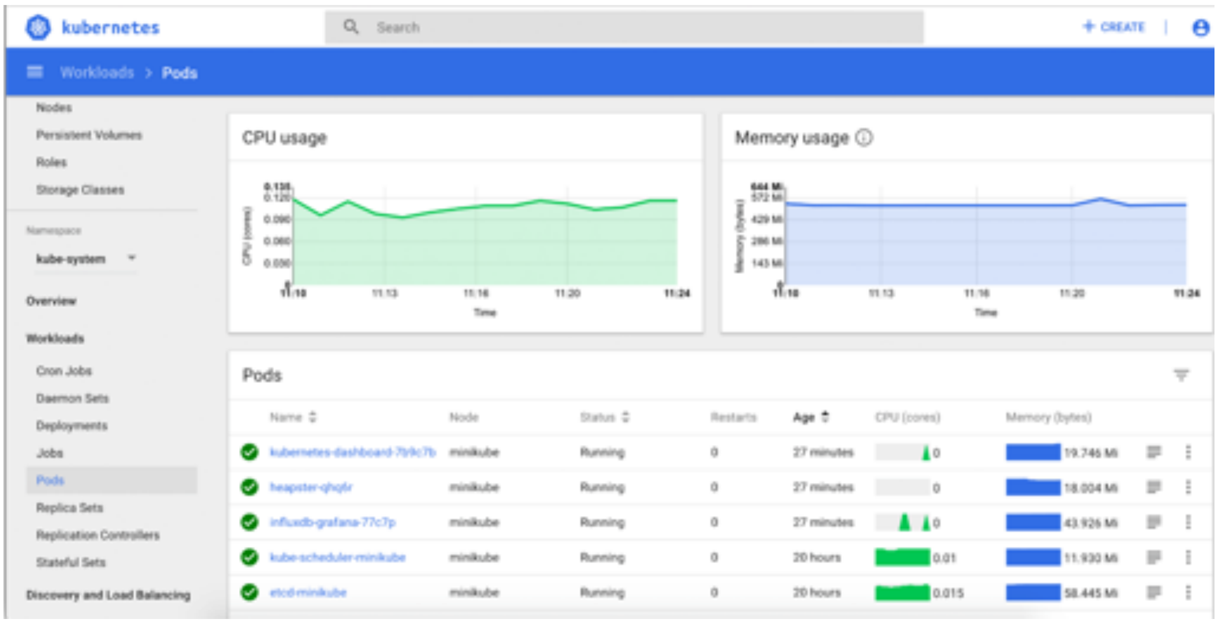
6.3 Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler (HPA) is a Kubernetes controller responsible to scale Pods according to the configured ranges. For the ELG, we plan to feed the HPA via a custom metric to allow for more fine-grained control of scaling (see Section 5).

6.4 Authentication and Authorization

In the context of this report, the terms *authentication* and *authorization* refer to access to the infrastructure components of the ELG (i.e., the Kubernetes cluster). In that regard, we exclude the actual access for users of the platform, because this is a matter of the application layer for which possible supporting services will be provided by the infrastructure.

Apart from the platform administrators, we may decide to give Language Technology service developers limited access to the cluster (e.g., to access logs and for debugging purposes), if at all – this remains to be evaluated and discussed in a later stage of the project. Kubernetes supports ABAC (attribute-based access control) as well as RBAC (role-based access control) modes. The Language Technology service developers will receive an access token that is limited to the services they develop themselves.



Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
kubernetes-dashboard-7b9c7b	minikube	Running	0	27 minutes	0	19.746 Mi
heapster-qhqfr	minikube	Running	0	27 minutes	0	18.004 Mi
influxdb-grafana-77c7p	minikube	Running	0	27 minutes	0	43.926 Mi
kube-scheduler-minikube	minikube	Running	0	20 hours	0.01	11.930 Mi
etcd-minikube	minikube	Running	0	20 hours	0.015	58.445 Mi

Figure 5: Kubernetes dashboard

6.5 Message Queue

As planned and described in Section 5.1, a message queue service will be deployed, allowing for asynchronous communication between Language Technology services and grid users. Details of the technology used (e.g., RabbitMQ, Apache Kafka) are not decided yet.

6.6 Backup

A backup solution will be available in later ELG infrastructure releases. For the development stage, no backup solution is planned. All parts necessary to set up the development platform anew, will be stored in GitLab and external image registries (see Section 7).

6.7 Helm

Helm is the standard package manager for Kubernetes. It allows rapid application deployment based on pre-made templates (Helm charts). Helm is intended to be used by the ELG administrators only, once sufficiently complex applications (e.g., ElasticSearch, Wordpress) need to be deployed to the platform.

There is also the possibility of configuring the whole ELG platform as a Helm chart itself, and this will likely be explored so that we are able to spin up multiple instances of the whole platform (e.g., when going from development stage to production stage).

6.8 Further Components (on demand)

Further components will be included in the platform when sufficient demand for them is encountered during development. This could include, e.g., a mail server to inform developers about failed deployments or other issues encountered.

7 Development Infrastructure

7.1 Repositories and Registries

To support the further development of ELG components, the tools described below have been set up.

7.1.1 ELG platform repository

The ELG platform Gitlab repository *gitlab.com/european-language-grid/platform/* will hold all code for the development of ELG internal components and related tooling. This will include, e.g., code for components like the service dispatcher and the service adapter.

7.1.2 ELG infrastructure repository

The ELG Infrastructure Gitlab repository *gitlab.com/european-language-grid/platform/infra* will hold all configuration files necessary to set up an ELG cluster. The configuration is expected to adhere to infrastructure-as-code standards. It will define:

- which infrastructure components are to be deployed
- which LT services are to be deployed
- all Kubernetes configuration files (e.g., for deployments and replica-sets)
- further configuration and parametrization

In the future, there will be either a parameterized version of these configuration files or separate repositories to hold the configuration for the development, integration and production clusters.

7.1.3 ELG LT service root

For the development of the LT services, GitLab groups have been set up, one for each consortium member. They can be found under the ELG root group at https://gitlab.com/european-language-grid/.

7.2 Continuous Integration

The general Continuous Integration workflow is based on:

1. One git repository for each service holding the source code for that service – it is maintained by the service developer. This is intended mostly for consortium partners; it is not mandatory to bring LT services into the grid.
2. One docker registry for each service holding the docker images for that service – it is maintained by the service developer. This can be either a registry associated with a repository under the ELG LT service root, or it can be an external registry. This is mandatory for all LT services to be hosted on the grid.
3. A set of runners (hosted by GitLab CI) to build the sources and push the docker image to the registry. This is optional, service developers not working on GitLab need to set up their own build servers.
4. One git repository (*european-language-grid/platform/infra*) holding the Kubernetes configuration files for all services to be deployed on the grid, which is maintained by the grid administrator.
5. Finally, a process for checking for and execution of updates (just ‘CI’ in the diagram in Figure 6), which is maintained by the grid administrator.

These five CI workflow components work together as shown in Figure 6.

This repository holding the Kubernetes configuration files will likely hold different configurations for each service, one each for the deployment on the development, integration and production cluster. The workflow starts with the service developer pushing a new commit to the respective git repository (in this case *gitlab.com/european-language-grid/example-service*). In GitLab terminology, *gitlab.com/european-language-grid* is a group, holding multiple repositories, one for each service. The group and a few demonstration services already exist and are connected to the second component, the docker image registry (*registry.gitlab.com/european-language-grid/example-service*). They are configured to automatically start a build process on GitLab’s shared CI runners, building the image file in the process. The image is configured to be uploaded to GitLab's docker registry.

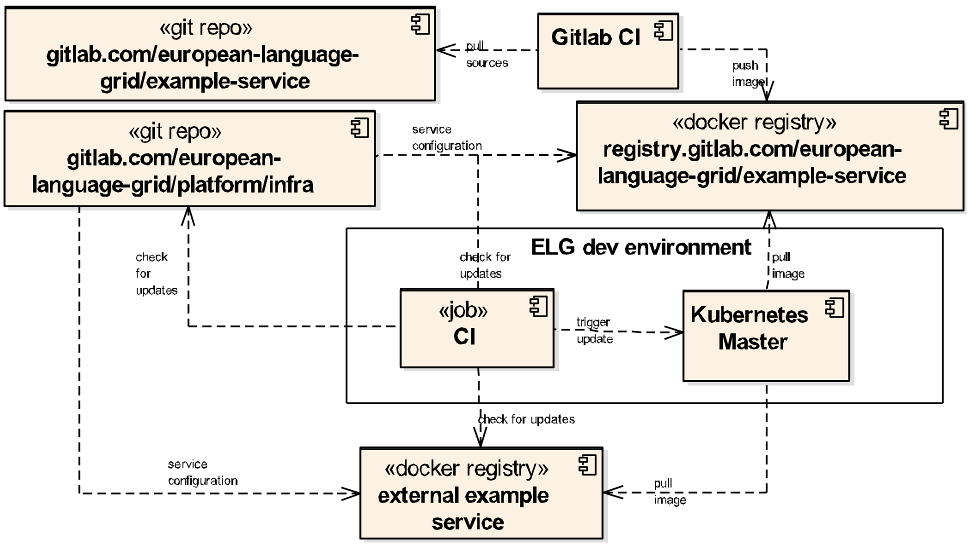


Figure 6: Overview of Continuous Integration

Apart from the image itself, various configuration files are necessary to specify how Kubernetes deploys the service. These are stored in a common git repository (*gitlab.com/european-language-grid/platform/infra*). Here, the following numbers need to be declared:

- How many nodes is the service to be deployed on?
- What are the resource requirements in terms of CPU/RAM?
- Which volumes to attach to the instances?
- Which version of the service to deploy?
- What is the scale up/scale down policy of the service?
- How is the service exposed (i.e., which URL, which topic[8])?
- others, as necessary

7.3 Support for Other Registries

Gitlab offers a convenient pricing policy for open source projects. Hosting the repository, the registry as well as using the GitLab CI runners is free of charge. As many of the services that will be deployed by the consortium members initially are expected to be open sourced, this was reason enough to choose GitLab over the alternatives (see https://about.gitlab.com/pricing/).

Nevertheless, realistically it cannot be expected that all services deployed in the ELG are hosted on GitLab (though GitLab offers support for private repositories and they are still free of charge, however, using the CI runners for private repositories is limited in usage without payment). Therefore, as shown in the overview diagram (Figure 6), it is possible to configure other repositories to be used to pull images from. These need to be configured inside the Kubernetes configuration alongside potential credentials to access the repositories (the

[8] In the context of message queues, a topic is a means of publishing information to a group of receivers who subscribe to the topic to receive all messages posted there.

credentials themselves must not be included in the configuration directly, but rather, e.g., indirectly via Kubernetes' 'secrets' mechanism). From the grid's perspective, there is no difference between the GitLab registry and external registries. Both are explicitly configured inside *gitlab.com/european-language-grid/platform/infra.*

7.4 Release Process

At the time of writing, the development cluster is up and running. Its release process is simpler than the one that will be implemented for the integration and production environments, and consists of the following steps:

- Initial configuration for the service in question: this means checking in the relevant Kubernetes configuration files to *gitlab.com/european-language-grid/platform/infra.* Here, the service version is configured with 'latest'. The initial configuration is expected to be created in close cooperation between the service developer and the grid administrator. This is a manual process that needs to be carried out only once, unless the service requirements change regarding the hardware equipment.
- Pushing a new version of the respective image to the registry.
- A to-be-configured process will be set up inside the grid to watch for updates to all registries. Kubernetes will create new pods automatically with the latest image version. Running containers will not be updated until they are terminated.
- Alternatively, pods can be shut down to enforce the use of the new image.

For the integration and production environments, a more controlled process is envisioned, where explicit versioning information is included in the Kubernetes configuration files. In these two clusters, the release process will adhere to standard ITIL-practices for management and documentation (i.e., change management, release and deployment management).

The integration and production clusters are not available yet. According to the project plan, they will be developed by 2020 and 2021, respectively.

8 Summary and Next Steps

Considering the requirements of geo-redundancy, compute centre clustering, consulting, flexibility of the CPU-vs-GB RAM ratio of the cluster instances, their expertise with Managed Kubernetes, and appropriate pricing, we selected SysEleven GmbH as the provider for the development stage of the European Language Grid. We designed an architecture based on state-of-the-art technology such as Kubernetes, the Horizontal Pod Autoscaler, and the Helm package manager. The architecture design further fulfils the requirements that we collected during the first phase of the project staying in constant contact with the consortium. We set up a development infrastructure using GitLab's Docker registry to store our Docker images and GitLab's repositories to store all other relevant code.