Building a Kubernetes infrastructure for CERN's Content Management Systems

Konstantinos Samaras-Tsakiris[1,*], *Rajula* Vineet Reddy[1], *Francisco* Borges Aurindo Barros[1,2], *Eduardo* Alvarez Fernandez[1], and *Andreas* Wagner[1]

[1]CERN
[2]Instituto Superior Técnico

Abstract. The infrastructure behind home.cern and 1000 other Drupal websites serves more than 15,000 unique visitors daily. To best serve the site owners, a small engineering team needs development speed to adapt to their evolving needs and operational velocity to troubleshoot emerging problems rapidly. We designed a new Web Frameworks platform by extending Kubernetes to replace the ageing physical infrastructure and reduce the dependency on homebrew components.
The new platform is modular, built around standard components and thus less complex to operate. Some requirements are covered solely by upstream open source projects, whereas others by components shared across CERN's web hosting platforms. We leverage the Operator framework and the Kubernetes API to get observability, policy enforcement, access control and auditing, and high availability for free. Thanks to containers and namespaces, websites are isolated. This isolation clarifies security boundaries and minimizes attack surface, while empowering site owners.
In this work we present the new system's open-source design contrasted with the one it replaces, demonstrating how we drastically reduced our technical debt.

1 Introduction

Google rewrites most of their software every few years [1]. Despite the cost, they consider it crucial to long-term success, because software requirements change as technologies evolve – and with them, user expectations. This practice typically reduces complexity and transfers knowledge into the new generation of engineers.

All these factors apply to CERN. Despite the much slower pace at which services evolve, CERN lives in the same dynamic technological environment. Without a constant input of effort, our software falls behind and fails to address modern expectations of features and aspects such as security, high availability, portability and isolation. At the same time unnecessary complexity accumulates: yesterday's custom solutions can often be replaced with new upstream components, the product of continuous standardization of solutions to problems that affect entire industries.

The increasing divergence between the original and present requirements results in *technical debt* [2]. The main purpose of this work is to *pay back technical debt* in CERN's Content Management Systems by modernizing the software architecture and making the service more secure and flexible (see section 4.1).

[*]e-mail: konstantinos.samaras-tsakiris@cern.ch

1.1 Why Kubernetes?

Kubernetes is for cloud native applications an extension of what the operating system is for traditional applications. It is becoming the de facto standard for Platform as a Service [3], abstracting computational infrastructure and standardizing deployment, so that an application can run unmodified on sites across the globe. Scientific applications are routinely deployed on Kubernetes [4–6], and even HPC use cases are being investigated [7].

At CERN the uptake is also evident. CERN IT has integrated Kubernetes in the Cloud Infrastructure and allows instant provisioning of new clusters. The ATLAS experiment is evaluating the replacement of all Grid computing services with Kubernetes clusters [8]. The Batch service, consisting of the largest portion of offline computational workloads at CERN, is prototyping a Kubernetes platform [9]. REANA, a system for Reproducible Analyses, is targeting Kubernetes as a main execution backend [10].

Many of these use cases are attracted by the promise of development velocity: rapidly shipping features, while maintaining highly available services [11]. A key element is to expand the pieces of a software stack that are immutable and versioned, declaratively configured, and self-healing.

Operator Pattern [12]

The Web Frameworks use case is not only about using Kubernetes as a deployment vehicle. Since we develop platforms and are concerned with their operational characteristics, we extend Kubernetes with custom APIs and controllers, building infrastructure management applications that are part of Kubernetes. Our applications range from integrating with other CERN systems to providing website management APIs and automating operations.

But what is included in the task of managing websites? The infrastructure, seen as an application, needs to have a concept of a website and let users define the website they need: its name, the technology, parameters, etc. Once a website is specified, the infrastructure needs to ensure that the website is automatically provisioned and set up. More than just a server, this task might include setting up storage and database, and integration with external CERN systems. After that, it needs to ensure that every component stays healthy and synchronized, propagating changes as requested by the user to every part.

In many cases, the solution has two components: a custom Kubernetes API and a controller that watches the API and ensures certain conditions. This pattern is called *Operator*, because it uses Kubernetes primitives to automate high-level operational workflows specific to the website technology.

1.2 Drupal at CERN

Drupal is an open-source *Content Management System* (CMS): a tool for site builders to organize and deliver content to their website visitors. It's used in 10% of the top-10k websites with the highest traffic [13, 14].

Who can benefit from a dedicated infrastructure for CMS websites? An organization that needs a dynamic Drupal environment, with high turnover of sites: universities, organizations comprising many departments and independent activities. Drupal is frequently embraced as an open source community-driven project [15], making it strategically attractive for *enterprise sustainability* [16]. Use cases range from simple blogs to professional newspaper publishing, from enterprise presentation to e-commerce, across government and private sector entities [17]. A frequently cited feature is the flexibility with which it adapts to bespoke requirements (custom modules), while scaling to large amounts of content and complicated editorial workflows. It has become the platform of choice for *public outreach*.

1.3 Article structure

Having laid down background information on the motivation, technologies and concepts used in this work, in the following sections we will describe:

Section 2	*What are the requirements* of Content Management as a Service at CERN?
Section 3	What system *currently* serves these requirements?
Section 4	What system did we design on Kubernetes *to replace section 3*?
Section 5	An experimental investigation of the new platform's efficiency
Section 6	Reflections on this work and plans for the future

2 Requirements for Content Management as a Service

The service supports website admins to host and administer Drupal websites directed to the grand public, such as experiment or departmental central websites. Some of the most popular sites based on this service are home.cern, atlas.cern, cms.cern, careers.cern and visit.cern. They form CERN's main outreach channel and are critical for the Organization's reputation.

Users of the service range across a wide spectrum of different professional profiles, and it's quite common that the responsibility of site building at CERN falls on administrative personnel, or personnel with little technical background in web technologies. This in turn shapes the kind of service we have to provide; it is, for example, impractical to rely on developer-centric workflows, like GitOps and CLI tools. A small fraction of our user base, however, indeed have web development experience.

The consequence is that the Content Management service has a dual mission:

1. to ensure the *high availability* and performance of these communication channels

2. to make site building and administration accessible to a wide-ranging user base, while remaining extensible for websites needing special features

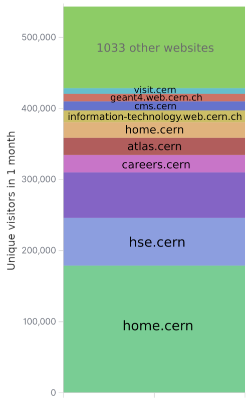
Control vs. customization

Curating the Drupal distribution, and critically, the application of *security updates*, is the responsibility of the infrastructure team. However, many websites need extra features and Drupal was selected exactly because of its extensibility (see section 1.2). Website admins should be able to use community modules, thereby extending Drupal specifically for their website – and assuming limited responsibility to keep custom code secure.

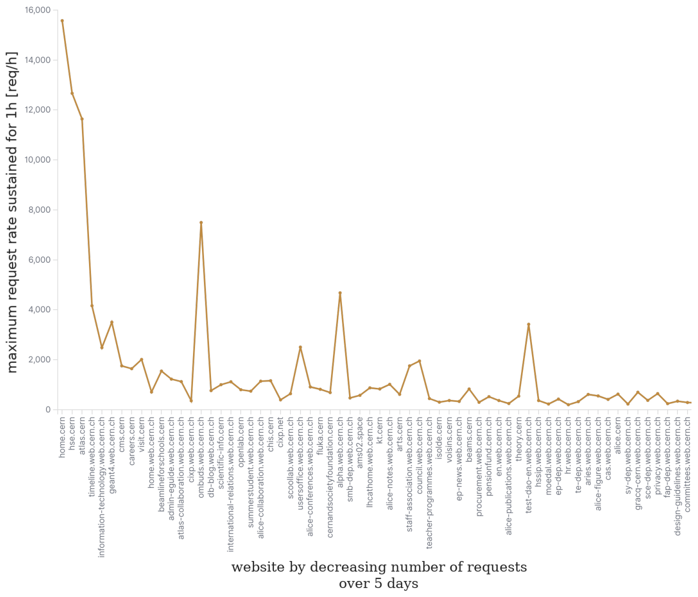
2.1 Load characteristics

home.cern is the most popular website at CERN, as shown in figure 1a. Out of *1043* Drupal websites currently hosted, it alone serves 32% of monthly unique visitors. The top 10 websites together serve 79% of all unique visitors, leaving only 1/5 of them headed for the other 1033 websites. This is an intrinsic characteristic of the service load, which is heavily skewed towards a very small number of critical websites.

Unique visitors over 1 month are taken as a measure of a site's popularity, or how much impact it has on the Organization's reputation, but a measure more suitable to assess an infrastructure on is the rate of HTTP requests. In section 5 we will describe an experiment on resource optimization in the Kubernetes infrastructure by assigning websites to different Quality of Service classes.



(a) *Public outreach*: The top 10 most popular Drupal websites are shown. home.cern appears twice as `home.cern` and `home.web.cern.ch`. `timeline.web.cern.ch` has machine traffic.



(b) *Maximum sustained throughput for high traffic websites*. The maximum throughput of the highest traffic websites was recorded over a period of 5 days. The websites with the highest traffic also have the highest throughput, showing that sustained bursts of traffic are uncommon.

Figure 1: Load characteristics

The 10 websites with the highest traffic are the target of 60% of all requests, and they have a high overlap with the most popular sites (fig. 1b). The most popular websites therefore, apart from the highest availability guarantees, need also the highest throughput.

What sustained rate of requests should a website be able to handle with stable response time? To better understand how the load impacts a single website (and therefore estimate the required hosting resources), we performed the measurements of figure 1b.

These observations align with expectations and requirements: critical websites should be able to handle a throughput of 30 requests per second with stable response times.

3 Current implementation

The infrastructure that currently serves the Drupal websites can be seen in figure 2. It runs on CentOS 7 and uses Puppet for configuration management. All servers run the same environment with systemd services, some of which are:

- HAProxy load balancer: routes requests to worker nodes, with an affinity cookie
- Keepalived: floating IP for the load balancer
- Apache httpd: serves Drupal PHP code, WebDAV interface and a few additional PHP management applications
- php-fpm: maintains a pool of worker processes that generate Drupal content

3.1 Request journey

The journey of an HTTP request can be seen in figure 2. Production websites respond to the load by spawning PHP workers, up to a maximum of 25. A worker process is always listening for requests, even without load. Test websites, on the other hand, spawn the first worker on demand and scale up to a maximum of 10 workers. The PHP memory limit for every website is 512MB.

3.2 Website isolation

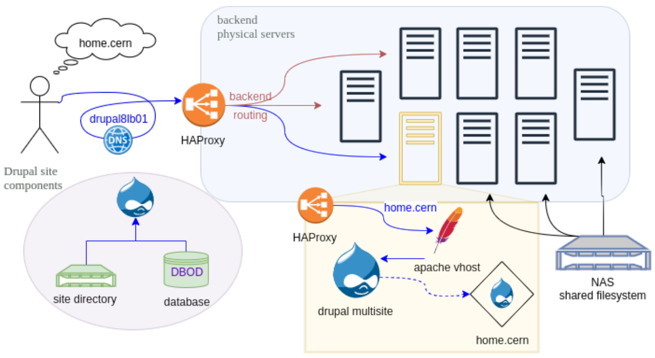


Figure 2: *Request journey in the physical infrastructure*. The infrastructure consists of 8 physical Linux servers behind a floating IP, equipped with NAS storage. The datapath to access a site is shown in blue. Drupal is configured in multisite mode to look up a separate directory and database for each site. The website's 2 data components: a persistent directory on the NAS and a database on an external service (DBOD).

Weak isolation is one of the biggest concerns of this infrastructure. Each website is assigned a Linux user. Its directory is owned by it and not accessible by the users of other websites. When Apache serves a request, it `chroots` the PHP process into the Drupal directory and sets the website's user. This distinction provides a basic isolation mechanism.

Nevertheless, websites are coupled in many places. We've never detected a cross-site security incident, but there are no `cgroup` limits to resources, and not enough security layers to defend against privilege escalation exploits. This is a critical concern, given the vulnerability of CMS software [18], and the impact that defacing a high-traffic public site would have to CERN's reputation.

A fundamental security practice is rapidly update upon security releases [19], but updating the multisite environment is fraught with dangers. All websites need to be updated in a massive, forced upgrade campaign, and if any has errors, it needs rapid debugging. Errors are supposed to be detected in a test environment, but not every website has one.

Furthermore, even though websites can be customized with contributed modules, there is no workflow to version control the websites.

3.3 Development workflow

Two types of website are supported administratively, corresponding to different Quality of Service (QoS) expectations: official (production) and test (test and development). There is no concept of *"environments"* or branches in this infrastructure. Developing a website involves maintaining a production website and one or more independent development websites. Data can be cloned between websites, so that the development website reproduce the production one.

A site admin that wants to safely develop a new content type or view, add a new module, or even change configuration, should start by cloning the production website to a development website. They need to keep track of changes, then reproduce them on the production website. There is no GitOps.

Despite seeming inefficient from a software engineering perspective, this workflow is acceptable by most website admins. The ones with development experience though would benefit significantly from version controlling configuration changes and extra modules.

3.4 Limitations of the current infrastructure

Reiterating the discussion, these are the major limitations of the current infrastructure. In section 4, the Kubernetes infrastructure lifts all of them.

- Hard to adapt resources, resulting in massive under-utilization
- Not very fault tolerant
- Weak site isolation increases the risk of severe security incidents affecting multiple sites
- Inflexible website environment limits development & testing workflows, makes upgrades cumbersome
- *Technical debt*: a lot of homebrew components built with legacy technologies specific to this system. Far from industry standards.

4 Pilot Kubernetes implementation

4.1 Common Kubernetes infrastructure for all Web Frameworks

Name	*Indicative use case*
EOS web hosting	Serve static HTML/CSS/JS/CGI stored on the EOS shared filesystem with simple deployment requirements
Platform as a Service (PaaS)	Custom web applications with flexible deployment on Openshift (Kubernetes)
CMS/Drupal	Websites for visual site building (see sec. 1.2)
Discourse	A home for communities around a common topic
TWiki	Wiki format content creation platform

Table 1: *Web Frameworks*: web hosting and content creation platforms based on different technologies. EOS web hosting and PaaS are *already using the common Kubernetes infrastructure*.

For each web framework there is currently a separate infrastructure, based on different technologies. This creates silos of operational expertise within the small engineering team that supports each one, which are *costly and inefficient* to keep alive in CERN's dynamic working environment. At the same time, there are a lot of software components developed in-house to support a single use case at the time, generating a large technical debt. Many requirements however are shared, such as interfacing with external CERN systems.

We therefore developed a common platform on the *Openshift Kubernetes community distribution (OKD 4)*, leaving only a thin business logic layer specific to each use case. OpenShift was chosen firstly for its production-tested multitenancy support, which we relied on for the present PaaS infrastructure. Openshift extends Kubernetes with tooling that simplifies our design [20]: a developer-focused console UI that we expose to end users, in-place upgrades of the control plane, node (machine) management API, monitoring and logging stacks.

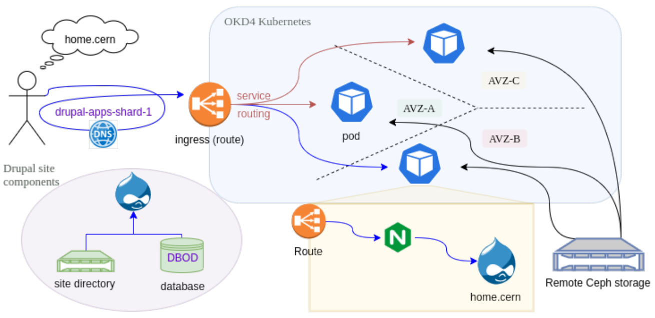


Figure 3: *Request journey through the Kubernetes infrastructure*. Contrast with figure 2.

The first web frameworks to use the new infrastructure are EOS web hosting (in production since *November 2020*, and PaaS (in production since *March 2021*). The Drupal use case is in Pilot phase, due to enter production in summer 2021.

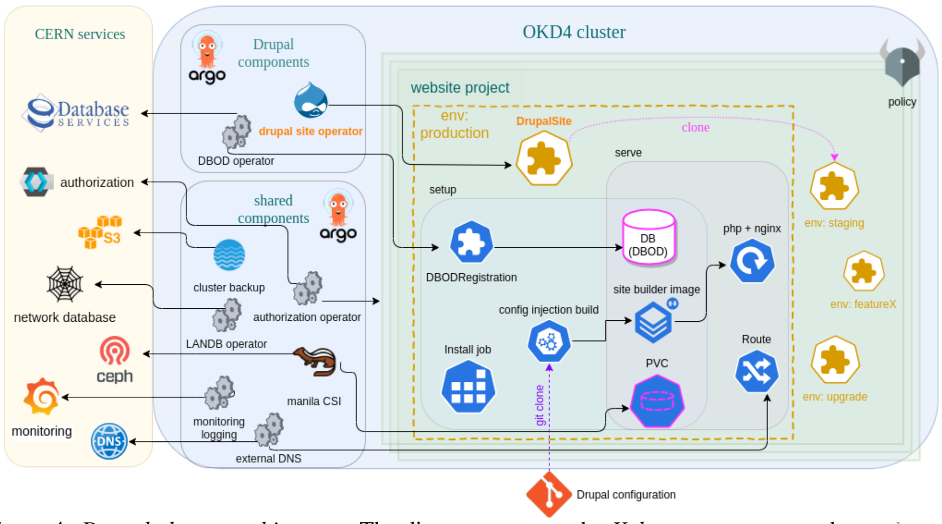


Figure 4: *Drupal cluster architecture*. The diagram presents the *Kubernetes resources* that make up a Drupal site, the controllers that manage them and the cluster infrastructure, and the external CERN systems with which the cluster integrates. Policy is handled with Open Policy Agent [21]. The DrupalSite CRD is an API for website definition. The components are deployed with Helm [22] and maintained with ArgoCD [23]

4.2 Serving perspective

The physical servers are replaced with virtual machines composing an OKD 4 cluster. Each website is served by 1 or multiple replicas of a pod with Nginx and PHP-FPM containers, in different cloud availability zones in case of critical websites. The server perspective can be seen by following an HTTP request in figure 3.

4.3 DrupalSite API: create and manage websites

The infrastructure can be seen as an application that offers its users functions to manage websites. It provides an API for website admins to specify the kind of website they need: what version of Drupal, what amount of resources, which git repository to fetch configuration from. Each `Project` (Kubernetes namespace) forms the administrative domain of a website: it serves a single production website. Website admins can similarly create different *environments* of their website in the same project for development or test purposes (which are full websites with independent data stores), clone data between websites, and take and restore backups. An overview of the components is in figure 4.

Following the operator pattern introduced in section 1.1, the business logic is implemented with the *`DrupalSite` Custom Resource Definition* (CRD) and the *drupalSite operator*. The `DrupalSite`, in turn, controls all the resources in the dashed box in the architecture diagram (figure 4).

Each website runs an immutable version of Drupal code, compiled from a version of the CERN Drupal distribution (controlled by the Infrastructure team) with a source-to-image build that injects user dependencies and configuration from a git repository. Site administrators with technical background thus gain extra flexibility in customizing their website.

We describe *more technical details of the Operator in [24].*

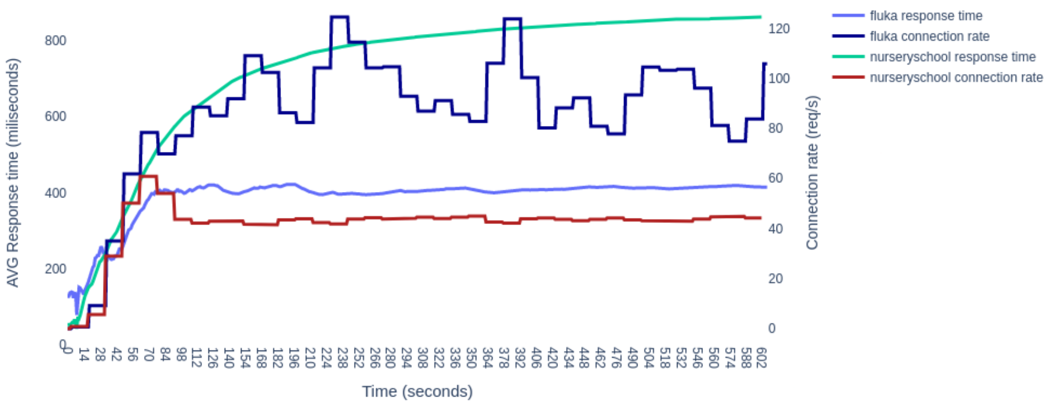


Figure 5: *Stress test for the Critical QoS*. The right Y-axis shows the load for `nurseryschool` and `fluka`. The left Y-axis shows the response time (average over simultaneous requests) for `nurseryschool` and `fluka`.

5 Measuring baseline resource requirements

The Kubernetes infrastructure can easily adjust its size according to the expected load. Each workload is also adaptable to load peaks: so far we're exploiting PHP-FPM's autoscaling capacity by scheduling server pods with fewer requirements than their memory limits (under the assumption that load is bursty and uncorrelated between websites) [1]. Nevertheless, we performed an experiment to measure the expected baseline resource consumption [2] and compare against the physical infrastructure, which, as we'll show, was overprovisioned.

We need a resource estimate to size the Kubernetes infrastructure so that it can handle the same baseline load as the physical infrastructure. To understand the baseline load and the resources needed to handle it, we define Quality of Service (QoS) classes based on required throughput that needs to be handled with a stable response latency. We perform a stress test to emulate the throughput and define the appropriate baseline resources.

5.1 Service level objectives

To define the load each QoS class needs to handle, we use the physical infrastructure as starting point (fig. 1b). Throughput peaked on the most popular website at around 16000 requests per hour, which averages on 4.4 requests per second. We define 3 QoS classes, which serve as Service Level Objectives (SLO), against the Service Level Indicator (SLI) of response latency under load. Because the latency depends on the complexity of each website, which is in the hands of the website admins and not the infrastructure, the SLO is met *not by defining a set value of the SLI, but by asserting that the SLI be stable* [3]. The 3 QoS classes are:

[1]So far we haven't found the Horizontal Pod Autoscaler useful, because the traffic on most "standard" QoS websites is too low to warrant multiple pods.

[2]The server pod's memory needs to satisfy each website's Quality of Service requirements. CPU in the CERN cloud is virtual and shared, so we don't take it into account when scheduling.

[3]Stability is interpreted as the latency settling to a fixed value after some time with steady load

- **Critical**: the most popular websites and therefore the most important to have high availability and request throughput (see section 2.1). They need to handle 30 requests per second (around 8 times the average on peak usage).
- **Standard**: these websites usually don't handle as much traffic and therefore don't need to have high request throughput. They need to handle 5 requests per second.
- **Test**: as in the name itself, these websites are used by website managers to test new features, and therefore are used by testers and developers. They only need to handle 1 request per second.

5.2 Stress test

The stress test consists of multiple simulated clients requesting URLs on the same website over a period of time. We copied a few websites (nursery and fluka used as examples) with varying content complexity from the physical infrastructure to experiment with on the Kubernetes infrastructure. The websites are stressed with load appropriate for the QoS class under investigation, and we tweak their configuration (mainly number of PHP workers), to get the lowest possible values for reasonable and stable response times.

The simulated clients live on a dedicated Kubernetes cluster that deploys a custom tool based on Locust [25] to make multiple requests to the targeted website on the new infrastructure. We spread the clients across multiple pods, each containing multiple processes that simulate users by requesting URLs at random [4].

website	Stress (req/s)	Response (ms)
nurseryschool	10	510
fluka	26	570

(a) Test

website	Stress (req/s)	Response (ms)
nurseryschool	18	240
fluka	48	103

(b) Standard

website	Stress (req/s)	Response (ms)
nurseryschool	41	870
fluka	74	400

(c) Critical

Table 2: *Stress test values for each QoS class*: max response latency and minimum load throughput.

Stress load

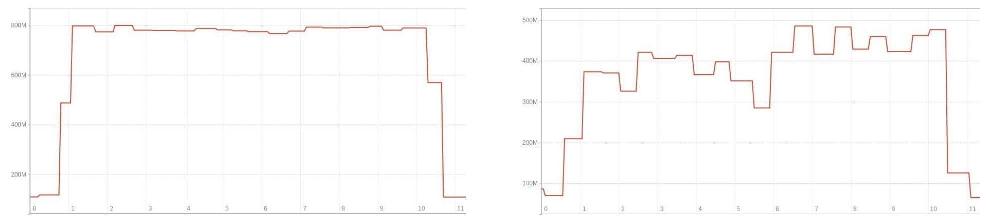
Multiple runs have been made with different configurations in order to find a suitable one for each QoS class to process the desired throughput with minimal resource consumption.

The throughput is affected by the load times on the website, meaning, a website with less content will take less time to handle a request and thefore the User will be able to do a new request faster than it would take on a website with more content.

Measurements

Figure 5 shows the load and the response during stress test for the evaluated websites. The stress tests ramp up during the first minute, after which they maintain the stress load for 9 more minutes. 10 minutes were sufficient for the response time to settle.

[4]First a URL discovery crawling of the website is performed, and only after having all the URLs, it picks URLs at random for the duration of the test



(a) nurseryschool (b) fluka

Figure 6: Memory use

The resource consumption was also monitored under load. The memory usage for nursery and fluka respectively can be seen in figure 6. Tables 2 show the highest response time under full stress and lowest requests per second for each QoS. Table 3 summarizes the test results in terms of resources needed by each QoS class under load.

5.3 Optimized resource allocation vs. physical infrastructure

Based on the results in section 5.2, we can see that response time may vary between QoS but remain under acceptable values. We can now estimate the baseline resources for the new infrastructure. The Expected Load is the maximum load registered during the tests plus 25% overhead.

QoS	**CPU**	**RAM(MiB)**
test	0.3	104
standard	2.3	257
critical	3	800

Table 3: *Peak resource consumption per QoS during stress tests*

The expected memory is:

$$\begin{aligned} TotalMem &= 1.25 * (C * L_c + S * L_s + T * L_t) \\ &= 20 * 1000MiB + 600 * 322MiB + 500 * 130MiB \\ &= 277750MiB \approx 336GB \end{aligned}$$

Where:

L_i = Expected max memory for each QoS class (table 3)
C = Total number of Critical Websites
S = Total number of Standard Websites
T = Total number of Test Websites

The physical infrastructure has 2TB of memory. To meet our SLO, the memory estimate for the Kubernetes infrastructure is 336GB. This is only *16.8% of the memory required in the physical infrastructure*, resulting in significant potential cost savings, even disregarding cluster autoscaling.

6 Reflections

It makes a big difference to discuss a design with 10 engineers rather than 3, and to have the peace of mind in case of emergency that many colleagues can take part. This is the hidden benefit of sharing a common platform. Especially in CERN's dynamic environment where the turnover of people is high, knowledge silos can be ill afforded.

The Pilot phase of this new infrastructure is still too immature to provide significant operational insights. We iterated on many design choices: the way to mark releases of the CERN

Drupal Distribution, the `DrupalSite` update logic, the Cephfs integration (remote storage)... Now we consider the design stable, while focusing development on Drupal-specific aspects, such as SSO integration. Functionality has been verified by copying and instantiating in Kubernetes many websites from the physical infrastructure, while the experiments of section 5 validate basic performance requirements. On the other hand, all the limitations of the physical infrastructure of section 3.4 have been lifted.

The next big challenge will be the production migrations in summer 2021. It is important to keep them as transparent to the website admins as possible and minimize disruption. From the power users however we expect the new features to receive a warm welcome and open new doors in their workflows.

Directions to explore

Develop once, run everywhere is a yet-to-be materialized promise. Plans for disaster recovery from a catastrophic failure of the CERN data center hinge on maintaining a public communications channel accessible. With Kubernetes cluster federation, using Public Cloud resources as a safety net is conceivable.

We will explore adding WordPress as a Service to the same infrastructure. Fundamentally, it should take no more than introducing a new build configuration.

The CNCF landscape [26] provides a salient overview of industry-standard solutions that can take this project the proverbial extra "mile". We plan to experiment with runtime security, root cause analysis, chaos engineering and serverless for the non-production environments. Kubernetes turns a homebrew system into a cosmopolitan denizen of the brave new world of the Cloud.

Acknowledgements

This work relies upon the contributions of all members of the Web Frameworks section of CERN's IT department, especially those that designed and implemented the common components. We thank specifically Alex Lossent, Ismael Posada Trobo, Joao Esteves Marcal, Iago Santos Pardo, Aleksandra Wardzinska, Michal Kolodziejski and Emmanouil Fokas.

References

[1] F. Henderson, *Software Engineering at Google* (2020), 1702.01715, http://arxiv.org/abs/1702.01715

[2] G. Fairbanks, *Ur-Technical Debt* (2020), ISSN 1937-4194

[3] N. Kaviani, D. Kalinin, M. Maximilien, *Towards Serverless as Commodity: A Case of Knative*, in *Proceedings of the 5th International Workshop on Serverless Computing* (Association for Computing Machinery, 2019), WOSC '19, pp. 13–18, ISBN 978-1-4503-7038-7

[4] C. Banek, A. Thornton, F. Economou, A. Fausti, K.S. Krughoff, J. Sick, *Why Is the LSST Science Platform Built on Kubernetes?* (2019), 1911.06404, http://arxiv.org/abs/1911.06404

[5] S. Hariri, M.C. Kind, *Batch and Online Anomaly Detection for Scientific Applications in a Kubernetes Environment*, in *Proceedings of the 9th Workshop on Scientific Cloud Computing* (Association for Computing Machinery, 2018), ScienceCloud'18, pp. 1–7, ISBN 978-1-4503-5863-7

[6] D.Y. Yuan, T. Wildish, *Bioinformatics Application with Kubeflow for Batch Processing in Clouds*, in *High Performance Computing*, edited by H. Jagode, H. Anzt, G. Juckeland, H. Ltaief (Springer International Publishing, 2020), Lecture Notes in Computer Science, pp. 355–367, ISBN 978-3-030-59851-8

[7] A. Beltre, P. Saha, M. Govindaraju, A. Younge, R. Grant, *Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms*, in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)* (2019)

[8] F.H.B. Megino, J.R. Albert, F. Berghaus, K. De, F. Lin, D. MacDonell, T. Maeno, R.B.D. Rocha, R. Seuster, R.P. Taylor et al., **245**, 07025 (2020)

[9] L.F. Alvarez, O. Datskova, B. Jones, G. McCance, **245**, 07048 (2020)

[10] T. Šimko, L. Heinrich, H. Hirvonsalo, D. Kousidis, D. Rodríguez, **214**, 06034 (2019)

[11] B. Burns, J. Beda, K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure* (O'Reilly Media, 2019)

[12] *Operator pattern*, https://kubernetes.io/docs/concepts/extend-kubernetes/operator (2021), accessed: 2021-06-03

[13] *Open Source Usage Distribution in the Top 10k Sites*, https://trends.builtwith.com/cms/open-source/traffic/Top-10k (2021), accessed: 2021-02-21

[14] *WordPress vs. Drupal usage statistics, February 2021*, https://w3techs.com/technologies/comparison/cm-drupal,cm-wordpress (2021), accessed: 2021-02-21

[15] D. Buytaert, *State of Drupal presentation (October 2019)*, https://dri.es/state-of-drupal-presentation-october-2019

[16] V. Brasseur, ed., *The Real Costs of Open Source Sustainability*, CERN Computing Seminar (2019)

[17] *Explore featured case studies*, https://www.drupal.org/case-studies (2021), accessed: 2021-02-21

[18] B. Shteiman, *Why CMS Platforms Are Breeding Security Vulnerabilities*, in *Network Security* (2014), Vol. 2014, pp. 7–9

[19] B. Csontos, I. Heckl, *Accessibility, Usability, and Security Evaluation of Hungarian Government Websites*, in *Universal Access in the Information Society* (2021)

[20] R. Jarvinen, *Extending Kubernetes with the Operator Pattern* (2019)

[21] *Open policy agent*, https://www.openpolicyagent.org/ (2021), accessed: 2021-06-03

[22] *Helm*, https://helm.sh/ (2021), accessed: 2021-06-03

[23] *Argo cd – declarative gitops cd for kubernetes*, https://argo-cd.readthedocs.io/en/stable/ (2021), accessed: 2021-06-03

[24] *Cern's 1500 drupal websites on kubernetes: Sailing with operators*, https://sched.co/iE362 (2021), accessed: 2021-06-03

[25] *Locust*, https://locust.io/ (2021), accessed: 2021-06-03

[26] *Cncf landscape*, https://landscape.cncf.io/ (2021), accessed: 2021-06-03