Enhancing the Availability of Web Services in the IoT-to-Edge-to-Cloud Compute Continuum: A WordPress Case Study

Gabriele Serra
Scuola Superiore Sant'Anna
Pisa, Italy
gabriele.serra@santannapisa.it

Pietro Fara
Scuola Superiore Sant'Anna
Pisa, Italy
pietro.fara@santannapisa.it

Daniel Casini
Scuola Superiore Sant'Anna
Pisa, Italy
daniel.casini@santannapisa.it

***Abstract*—The IoT-to-Edge-to-Cloud compute continuum presents vast opportunities for innovative applications, including crowdsensing, which leverages interconnected devices to gather real-time data. In domains like autonomous driving, crowdsensing enables traffic information sharing through web services. In this context, web services, like those based on Content Management Systems (CMS), are often used by drivers and passengers to share data about user experience, traffic congestion, and high-definition maps. However, ensuring high availability becomes crucial to maintain accessibility and reliability against usage peaks. This paper proposes a modern WordPress deployment approach that takes advantage of cloud-based to realize a cost-effective horizontal scalable architecture, leveraging Amazon AWS. The architecture suggested was implemented to test the effectiveness and released as a set of architecture-ready-to-use templates. Experimental results are provided to measure per-request response times under different autoscaling policies and bootstrap times.**

I. Introduction

In the realm of the IoT-to-Edge-to-Cloud compute continuum, the convergence of technology has opened up new possibilities for innovative applications. One such application is crowdsensing, which harnesses the collective intelligence of individuals and their interconnected devices to gather and analyze real-time data. For example, crowdsensing plays a crucial role in autonomous driving, allowing drivers and passengers to share information about traffic conditions using web services such as those commonly leveraging Content Management Systems (CMS). Improving the availability of IoT-to-Edge-to-Cloud applications is an open challenge that is increasingly addressed using cloud computing. Indeed, in this scenario, ensuring the high availability of web services becomes paramount as peaks in usage could potentially hamper accessibility and reliability.

High availability of web services refers to the ability of a system or application to remain operational and accessible, even during periods of high demand or potential disruptions. In the context of autonomous driving, where crowdsensing enables the exchange of traffic information among drivers and passengers, web services are essential for live data sharing and informed decision-making. One of the leading examples is represented by Waze[1], a free mobile street navigation application that heavily uses crowd-sourcing assistance from community users and mobile data analysis [1]. Waze uses a Content Management System (CMS) to collect user feedback and suggest the shortest path to the destination.

Another example, still from the autonomous driving domain, is updating High-Definition maps (HD maps). Indeed, modern autonomous driving software stacks (such as, for example, Apollo [2], [3]) use such maps for having advanced knowledge about the roads, e.g., for having precise information about where a traffic light should be located. As road conditions change over time, these maps must be constantly updated, mainly using crowdsensing [4], which relies on cloud services. Again, here availability is important to ensure users always have updated maps.

As a practical use case of the cloud-based chunks of a massively distributed IoT-to-Edge-to-Cloud application, this paper focuses on the availability of the popular WordPress service.

WordPress is one of the first choices when dealing with dynamic web content. WordPress is an open-source content management system built over PHP and MySQL started in 2003. Its modular architecture allows developers to customize it in remarkable ways. WordPress is the platform of choice for over 39% of all sites across the web [5], also thanks to its permissive GPL2 license. In 2003, when the first version of WordPress was released, it was not built to sustain the number of users that, nowadays, could potentially visit a website simultaneously. Furthermore, it was not designed to take advantage of all modern solutions, such as object caching and database replication. Typically, WordPress is deployed without relying on any modern, scalable cloud technology, on a physical machine or, most of the time, on a virtual machine in a shared server. However, it is worth noting that the proposed approach is not limited to WordPress and can be applied to any general large web application.

Contribution. This paper proposes a modern WordPress deployment that uses cloud-based instruments to address the

[1]https://www.waze.com

mentioned issues. In particular, the paper aims to help web-hosting services or cloud-migration companies to build a robust, high-available network of WordPress instances. To this end, we developed an architecture that uses Amazon AWS services. The approach developed in this paper could be implemented using different cloud provider services such as Microsoft Azure or open-source Open Stack. We also provided an architecture implementation and realized it through architecture templates. Efficiently exploiting cloud services and infrastructure requires technical expertise and a deep understanding of the cloud provider-offered services. Therefore, we strongly believe a set of documented predefined templates can represent a starting point for companies or developers with limited prior experience to understand how effectively deploy a large web application, drastically improving the availability of any IoT-to-Edge-to-Cloud applications.

II. Related work

The basic principles in scaling up a WordPress instance are the same for scaling up any sizeable web application. Due to the number of active WordPress instances, the community often debates how effectively to scale up a WordPress-based web application horizontally. Many companies offer WordPress horizontal scaling plans [6], [7], [8]. Further, research papers and tech reports provide solutions for augmenting WordPress horizontal scalability, typically achieved by adding more machines to the allocated pool of resources. [9] represents a noteworthy solution, which reports some takeaways concerning developing a network of approximately half a million blogs. However, the solutions mentioned do not offer a detailed architecture design or documentation. Instead, they offer mainly a list of tips or, in other cases, expensive hosting plans.

The most scalable solution of WordPress multisite instance is represented by *WordPress.com*. They decided to publish the plugin used in their production version of the architecture, HyperDB. HyperDB abstracts access to geo-distributed databases[2]. However, they did not release any further documentation regarding the architecture.

To the best of our knowledge, the most detailed and comprehensive work was published by Amazon [10]. Our proposed architecture is based on the mentioned work and extends it. The differences between our proposed architecture and the one proposed by Amazon consist mainly of

- replacing the synchronized file system with a stateless block file system,
- making use of state-of-the-art packages,
- providing benchmarks showing web server response times under heavy-loaded situations,
- providing a build-and-validation template mechanism;

The proposed approach can work well in a heavy-load scenario for all the reasons mentioned in the introduction. Simultaneously, a single WordPress instance can be reasonably scaled vertically, i.e., by adding more computational power to the existing machine, arguably representing a less expensive choice.

The architecture provided by Amazon suffers a high-performance penalty when serving highly-dynamic pages. In Amazon's proposed approach, each compute instance shares a network synchronized file system (EFS) mounted as NFS. Network file systems are charged for each byte of data retrieved; thus, there are more cost-effective solutions than this. Further, EFS is subject to throttled burst with a limit depending on the file-system size. Their proposed approach injected fake data to increment the file system's size and get a greater burst limit, paying a higher cost.

Conversely, our benchmark showed that a batch of I/O operations (such as reading, writing, and listing metadata) in EFS could experience a latency that is 10x higher than on a block file system. Previous works, such as [11], confirm our measures and discuss how to tune the environment to get the best performance which is, unfortunately, not comparable to work with a local block file system. To confirm our conclusion, in [12], the author analyzes the burst throughput limit in-depth, highlighting that it is not even an option to serve a PHP application with thousands of files using a network file system.

Several contributions have also been presented in the academic research domain. For example, a probabilistic approach has been proposed for reducing physical computational resources requirements in an elastic cloud computing environment [13]: basically, an admission control decides if a new service can be admitted in a system reducing the risk of failure of already deployed ones. Bowers et al. [14] presented HAIL: high availability and integrity layer to guarantee the integrity and high availability in cloud storage. Another high-availability architecture for IoT-to-cloud services has been proposed in [15]. Kanso et al. [16] presented an approach for enforcing high availability acting at the application level. Casini et al. [17] considered the local load balancing of real-time applications running on multicore embedded systems without targeting a distributed system.

Overall, the proposed architecture differentiates from the others because it presents an open ready-made approach, considers deploying a WordPress instance, and overpasses performance limits using a local block file system.

III. Architecture

Typically, highly available web applications are organized using the three-tier architecture. The three-tier architecture is a software architecture that forces an application's organization into three logical tiers: the presentation tier, or user interface; the application tier, where data is processed; and the data tier, where the data associated with the application is stored.

The three-tier architecture and its instantiation in the context of this work are shown in fig. 1.

The main benefit of adopting this architecture is that each tier can be updated or scaled as needed without impacting the other tiers. Today, most three-tier applications are targets for

[2]The plugin is available at https://wordpress.org/plugins/hyperdb/

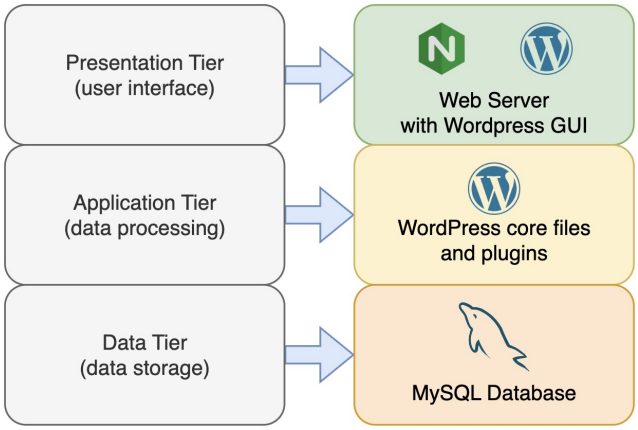

Fig. 1. Three-tier cloud architecture and its instantiation in the context of the WordPress Case Study.

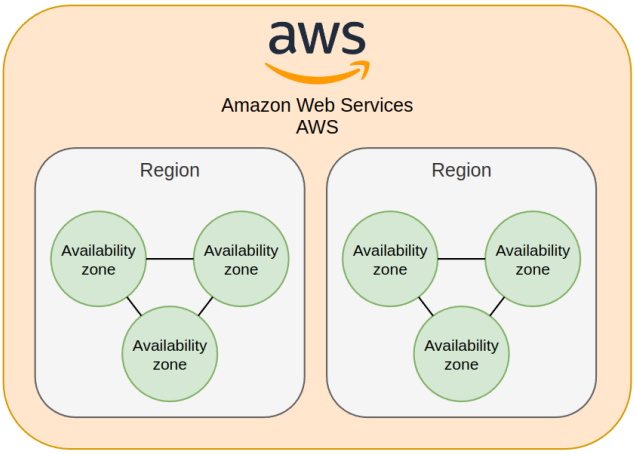

Fig. 2. Multiple Availability Zones (AZ) in a region.

modernization, using cloud-native technologies such as containers and microservices, and for migration to the cloud [18].

In our WordPress environment, the tiers are, respectively:

- the web server that provides the user interface (both end-users and administrator interfaces) and static assets;
- the application logic composed of WordPress core files and plugins;
- the database server hosting end-user sessions and data.

To let the WordPress instance scale both horizontally and vertically, all the tiers must be designed to support scalability. In the following subsections, we analyze deeper how to organize the architecture accurately for all tiers.

A. Make the web-tier scalable

1) Compute machines: All major cloud providers do not offer ready-made web hosting. Instead, they offer re-sizable virtual machines. Typically, developers can customize virtual-machine instances to fit their needs regarding performance and cost.

When deploying a computing instance, the system administrator must specify the region where the instance will run and the respective availability zones (AZs), i.e., logical data centers located within a region available for use by any customer. Each region is designed to be isolated from the other regions hence resources across different regions are not replicated. However, it is possible to distribute instances across multiple AZs so that if an instance fails, another one in the same region but in a different AZ can handle requests. The Figure 2 shows how multiple AZs are organized in a region.

2) Resources auto-scaling: As pointed out in the introduction section, WordPress instances that do not use modern cloud services must be redesigned, taking into account the traffic peak. Elasticity is one of the most exciting characteristics of cloud services. Indeed, the cloud provider could provide more computing capacity to the application, instancing more VMs, and making the application scale horizontally. The provisioning could be automatized according to traffic conditions without the need for manual intervention. Our architecture configures the computing machines' number adding one unit during traffic spikes to maintain performance requirements. When the traffic decreases, the auto-scaling functionality automatically resizes the group to reduce costs. Indeed, cloud providers charge customers per hour of computing machine usage. As demonstrated in Section V, the number of available computing units can follow target performance. Auto-scaling ensures that only the necessary computing units are instantiated in a given time window. This means that you can avoid over-provisioning your resources, which can lead to unnecessary costs.

3) Load-balancer: Our architecture supports dynamic addition and removal of compute instances. Furthermore, instances are deployed over different AZs in a given region. Instances are always available; therefore, we included a load-balancing solution to distribute end-user requests. The load balancer could be configured to distribute requests across different EC2 instances. AWS offers different kinds of load balancers; however, the suggested one to implement HTTP/HTTPS balancing is called *Application Load Balancer*. The application load balancer can be customized to balance application load using HTTP packet headers. Furthermore, it can also be configured to perform application health checks and machine health checks. In our implementation (refer to Section IV), we configured a health check on the `/wp-admin` dashboard. The load-balancer requests the admin dashboard every interval of seconds, and then, if it does not receive a reply from an instance, it stops sending requests to the instance and signaling it as faulty. You can customize health checks to your specific use case.

B. Make the application-tier scalable

Our architecture uses multiple web servers in automatic scaling configuration to maximize availability and minimize costs. However, to let the application instance scale on different machines means the application must be *stateless*. A

stateless application does not need knowledge of previous interactions and does not need to store data.

In the context of web applications, an application is said to be stateless when all end-users receive the same response when making the same request, regardless of which web server (when more than one) processed their request. Consequently, A stateless application can scale horizontally since any web server can service any request.

Further, each web server instance does not need to be aware of others' presence, and the only requirement is to distribute the workload across different peers.

By its nature, WordPress is partially stateless; it relies on cookies stored in the client's web browser and mainly stores data (such as posts and user info) in the database. However, WordPress was initially designed to run on a single server. As a result, it stores some data on the server's local file system. When running WordPress in a multi-server configuration, saving data on the server's local file system creates inconsistencies across web servers. For example, if a user uploads a new image, it is only stored on one of the servers. This demonstrates why the default WordPress running configuration needs to be augmented to allow moving data out of the application tier, thus preserving a stateless behavior.

In our approach, we decided to make the WordPress core completely stateless. We distinguish the files that need to be moved out of the WordPress application tier as log files and media. Media files are static assets, so we made them available through a file bucket. Regarding logs, we created a shared file system, synchronized it among all compute instances, and mounted it at boot time.

C. Make the data-tier scalable

When using WordPress, the database is one of the most critical components that affect performance. Therefore, as pointed out in previous sections, it is the only stateful component. Then the database cannot be hosted directly with web servers on computation instances, but it must be hosted on a reliable, always-available node. However, tuning up a replicated database with all the implications can be challenging. Consequently, we decided to opt in for a managed database. A managed database is a cloud computing service handled directly by the cloud computing company. Unlike self-deployment databases, developers do not need to set up nor maintain the service; instead, it is up to the provider to oversee the database's infrastructure. A managed MySQL-compatible database is transparent to the application that can interact using SQL without worrying about consistency and replication. Managed databases also automatically handle failovers. Furthermore, to reduce the workload on the database, we decided to cache frequent queries.

1) Database caching: Database caching is a common practice when applications are read-heavy such as WordPress. The results of selection queries are stored in a low-latency memory (typically in a key-value structure). A variety of object caching systems is available nowadays. One of the most used is Memcached, an in-memory key-value store for small chunks of data. As the database scales, the cache must consequently scale up; thus, a suitable option is taking advantage of a managed cache service.

IV. Implementation

Next, we discuss how the proposed architecture has been implemented. First, we present the AWS services used. Second, we discuss the WordPress network management details of the proposed solution. Finally, we discuss the deployment of the solution.

A. Involved AWS services

The architecture described in Section III was implemented, making use of Amazon AWS services. We decided to instantiate all needed resources in an Amazon Virtual Private Cloud environment (VPC). A VPC is a logically isolated virtual network where the solution architect has complete control over the networking environment. A VPC can also be subdivided into subnets reachable employing route tables and network gateways.

Then, our reference implementation spawns all resources in a newly created VPC, deployed in the *eu-west-1* region, physically located in Dublin, Ireland. The region mentioned above is the region with the lowest price; therefore, it is a common choice when dealing with an application that serves the European market. The VPC works across two (out of three) availability zones (AZs) to maximize availability and fault tolerance.

A content-delivery network node represents the edge of the network, realized using AWS CloudFront. The goal of Cloudfront is to distribute our service in a geographic location nearer to the end-users. In our particular scenario, the most requested static assets will be cached and served from the AWS Datacenter in Milan. Using a CDN, we can deploy our application in a specific geographic location (such as *eu-west-1*) without worrying about the latency experienced by end-users. AWS Cloudfront is configured with two different origins, an Application Load Balancer and an S3 bucket to serve images.

The Cloudfront endpoint is linked with the primary domain name. Each site of the network has a third-level domain or can be mapped to a second-level domain. The entire DNS management is done using AWS Route 53, a highly available and scalable cloud Domain Name System (DNS) web service.

As pointed out in Section III, all media, such as photos and videos, are not directly served by the web server. Indeed, these static assets are placed into an S3 bucket that is shared among all webserver instances. Amazon S3 (in which S3 stands for Simple Storage Service) is a low-cost object storage service that serves static assets. Using Amazon S3, our web servers do not need to serve all static files; thus, they can generate dynamic content for end-users.

Instead, the Application Load balancer tries to distribute all requests from end-users to computing instances hosting our web servers. The load-balancer distributes requests balancing the load on each instance, and performs an instance health

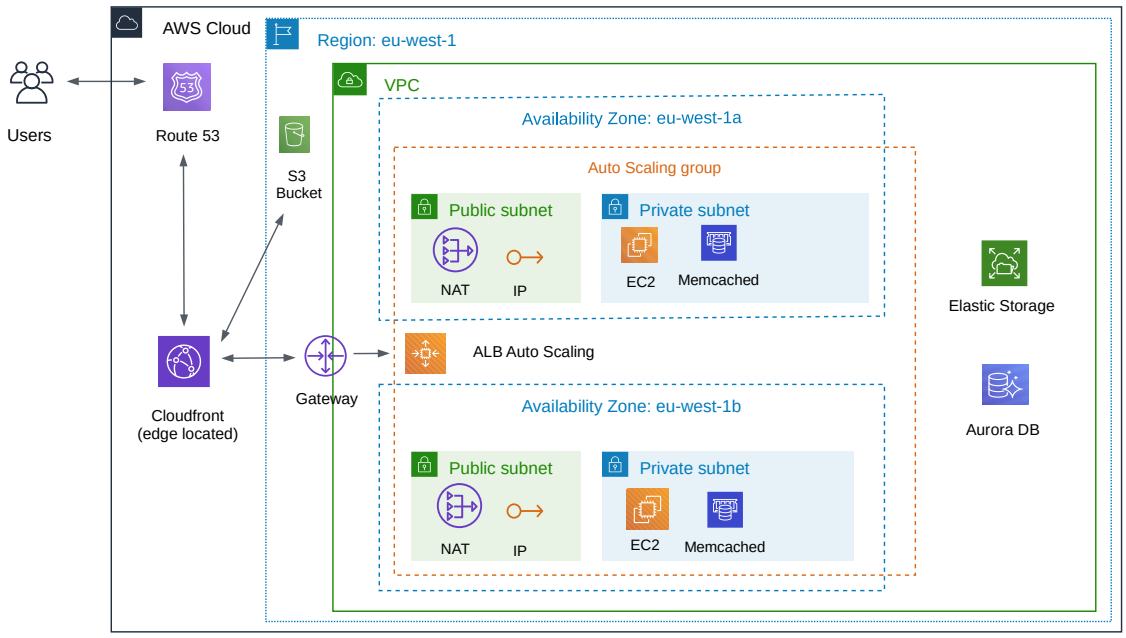

Fig. 3. Implementation of the proposed architecture using AWS services.

check. If an instance does not reply for any reason, the load balancer can replace the instance by performing a hot swap. Further, the load balancer can spawn additional computing instances.

Computing instances are implemented using Amazon Elastic Compute Cloud (EC2) instances. EC2 instances are customizable virtual machines that provide computing resources at a fixed hourly cost. Webservers, then, are deployed over an EC2. Each EC2 uses a private block file system, thus executing the stateless WordPress core.

Posts, user credentials, and products, all users' session information, are saved onto an AWS Aurora database. Amazon Aurora is a MySQL-compatible relational database built for the cloud. Amazon Aurora features a distributed, fault-tolerant storage system that auto-scales up to 128TB per database instance. It delivers high performance and availability with up to 15 low-latency read replicas and replication across three Availability Zones (AZs). One of the most excellent features is that the replication is transparent to the application, which works out of the box without installing additional plugins or components. We chose a Master-Slave configuration. Also, we decided to cache the most recent queries using an AWS Elastic Cache instance. The support is provided by AWS and compiled by us for PHP 7.4. The plugin W3 Total Cache is used to make everything work smoothly.

Our reference architecture was implemented using Amazon AWS Cloudformation templates. While most users use the AWS Web interface for their everyday monitoring activity, it is difficult to version and track architecture changes without a versioning tool. Therefore, AWS CloudFormation allows you to model your architecture using *json* or *yaml* templates, treating your infrastructure as code. Cloudformation templates describe the resource and their dependencies, so it is possible to create, update, provision, and delete the entire stack of resources instead of manually managing each of them.

B. Wordpress network management details

A super-admin user (network manager) can create new sites and manage the network entirely. The network manager decides if end-users can create new sites.

All sites share templates and plugins. From the "network panel", the Super Admin can enable/disable plugins and templates; this is a smart way to install/update/enable a plugin in all the networks.

The database is unique for the entire network and contains two kinds of tables: shared and private. Shared tables have information common among all sites of the network, such as users of the network. Private tables, instead, contain information referring to a single site. Each private table has a unique prefix that makes use of the blog-id number.

The system administrator can use the WP CLI. To specify the target site, you need the site URL, i.e.

```
wp command --url=sub.example.com [opt]
```

We decided to make the block file system not writable by the WordPress instance. In this way, when a plugin tries to write the file system, an error will pop up, enhancing the system's security degree.

C. Deployment

The repository provided contains a set of nested templates which are run in order from the master template. Running the master template, you can deploy the entire stack. Furthermore,

```
# Master template
Resources:
  # ...
  web:
    Condition: AvailableAWSRegion
    DependsOn: [
        efsfilesystem,
        vpc,
        publicalb,
        securitygroups,
        cloudfront ]
    Type: AWS::CloudFormation::Stack
    Properties:
      Parameters:
        DatabaseClusterEndpointAddress:
          !GetAtt [
            rds,
            Outputs.DatabaseClusterEndpointAddress ]
        # ...
        WebAsgMax:
          !Ref WebAsgMax
        WebAsgMin:
          !Ref WebAsgMin
        WebInstanceType:
          !Ref WebInstanceType
        WebSecurityGroup:
          !GetAtt [
            securitygroups,
            Outputs.WebSecurityGroup ]
        # ...
      TemplateURL: 04-web.yaml
  # ..
  cloudfront:
    Condition: DeployCloudFront
    DependsOn: [ publicalb ]
    Type: AWS::CloudFormation::Stack
    Properties:
      Parameters:
        CloudFrontAcmCertificate:
          !Ref CloudFrontAcmCertificate
        PublicAlbDnsName:
          !GetAtt [
            publicalb,
            Outputs.PublicAlbDnsName ]
        WPDomainName:
          !Ref WPDomainName
      TemplateURL: 04-cloudfront.yaml
  # ...
```

Fig. 4. Example of template code.

you can run templates individually (to debug, for instance), providing the required parameters at each step.

The master template, shown in fig. 4, can be launched by taking advantage of AWS CLI, using the `aws cloudformation create-stack` command, and passing the master template.

V. Experimental Evaluation

This section presents our experimental evaluation campaign to validate the implemented approach. The experimental campaign targets the web application's *response time* and the *bootstrap time* of each architecture component.

A. Response time

The steady-state phase of the campaign evaluates the system's resiliency and capacity to sustain an increasing number of requests.

Our experimental evaluation was carried out by increasing the number of concurrent requests, performing a `GET` request, and tracking the response time of the webserver in generating the content.

The application load balancer was configured to equally distribute the load over available targets. The auto-scaling groups were configured to scale the capacity as traffic changes occur dynamically.

We considered two different dynamic autoscaling policies:

- `Target tracking scaling`: increase/decrease the current capacity based on a specific monitored metric;
- `Step scaling`: increase/decrease the current capacity based on step adjustments, configured using monitoring alarms.

Next, we briefly discuss the main characteristics of the two policies.

Target tracking scaling. The target tracking scaling specifies a target value for a specific metric and aims at maintaining the target value by minimizing the error. The autoscaling policy can be configured with different metrics [19], such as the average CPU utilization, the average number of bytes received/transmitted by a single instance on all network instances, and the average application load balancer request count per target. In this experiment, we consider both the average CPU utilization of 50% and an average request count target of 1000 requests as metrics. Indeed, for example, meeting an average CPU utilization of 50% can make the system robust to handle traffic spikes without maintaining excessive idle resources. The dynamic autoscaling policy then scales the number of instances to keep the CPU utilization value around 50%.

Step scaling. The step scaling policy considers the same metrics as the target tracking scaling. However, it takes a different input: an upper and a lower bound relative to the breach threshold, i.e., the threshold that triggers an autoscaling action when overcome. Most commonly, the autoscaling action consists in adding or removing one virtual machine, but also more complex policies exist [20]. In AWS, this is achieved by means of CloudWatch alarms [21]. Also, we considered the average CPU utilization as a metric in this configuration. The step scaler is set configured with two thresholds: 45% and 55%. The autoscaling action consists of adding or removing *one* virtual machine when the threshold is surpassed. The same action is taken when the scaling policy is configured to meet an average request count of 1000.

Experiment. This experiment consists in performing concurrent requests to the web server by numerous clients. The number of concurrent requests ranges N ranges from 1 to 10000. The application load balancer was configured with the mentioned scaling policies.

The same metric was selected for each scaling policy, and we performed two sets of experiments using the average CPU utilization and the number of requests per target as a metric.

The computing instance hosting the web server is a `t2.micro` computing instance. The minimum number of

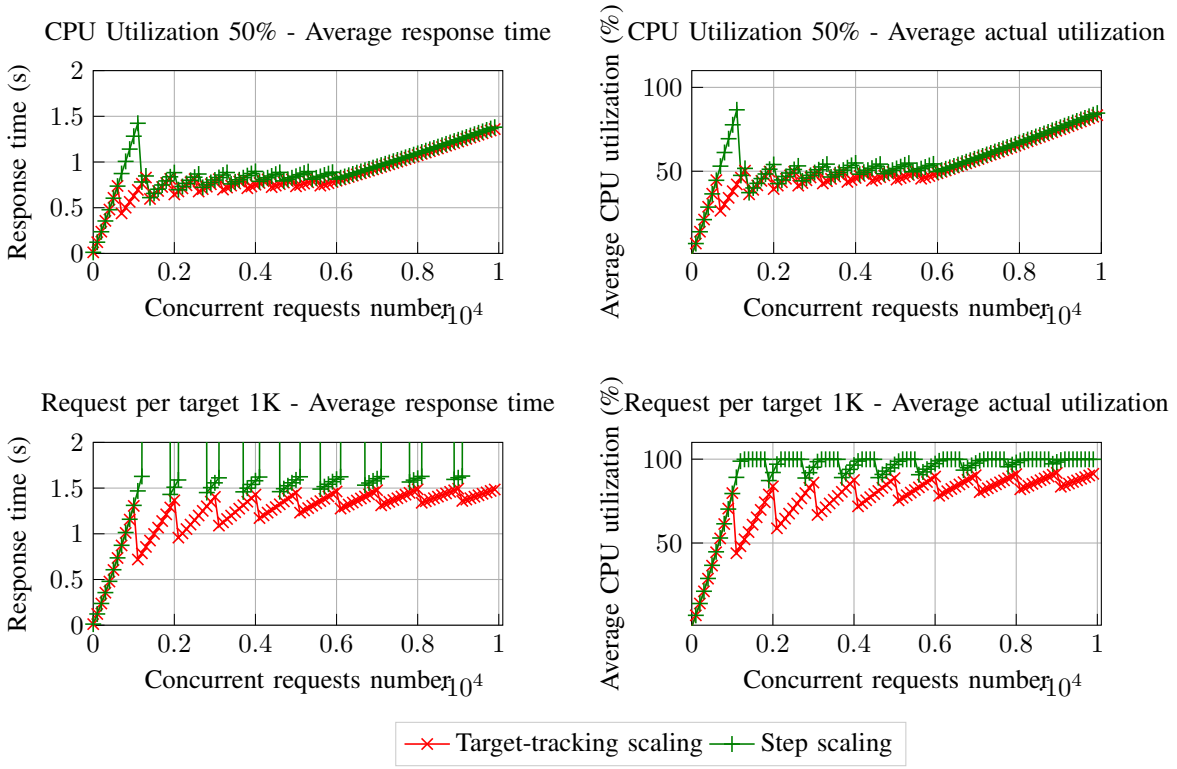

Fig. 5. Response times and average CPU utilization when considering a CPU utilization of 50% as a metric for the dynamic autoscaling (first row) and 1000 request for target (second row).

computing instances was set to 1, and the maximum number to 10.

Results (Figure 5) show that the application load balancer, in both cases, can honor the requirements.

Both scaling policies exhibit similar behavior in the first set of experiments (the first row of Figure 5). The average CPU utilization remains constant and near 50%. When the number of requests grows above 6000, the load balancer stops launching computing instances due to the maximum constraint specified.

In the second set of experiments (second row of Figure 5), scaling policies exhibit still a similar behavior but, due to the intrinsic difference in the control policy, they reach different results. Due to the presence of the threshold, the actuation of the scaling happens with a delay with respect to target tracking scaling, bringing the system to an overloaded status (the average capacity reaches 100%) and therefore bringing it to an unresponsive web server.

B. Bootstrap time

In this experiment, we measured the time needed to set up the entire architecture since the deployment of templates, i.e., the so-called *boostrap time*. Since templates were developed, setting up the entire network stack requires only deploying them using the AWS CloudFormation service.

Bootstrap time is important for maintenance, as it provides important information about how long maintenance windows must be. Further, the bootstrap time of the computing nodes gives fundamental hints about computing instance warm-up and cooling-down times.

Warm-up and cooling-down times are used as delays in actuating control actions to reach a scaling convergence. Until its specified warm-up time has expired, an instance is not counted toward the aggregated instance metrics of the auto-scaling group. If the group scales out again, the instances that are still warming up are counted as part of the desired capacity for the next scale-out activity. The intention is to continuously (but not excessively) scale out.

Table I shows the time (seconds) needed for each template from the deployment to the actual availability of resources.

VI. Conclusions & Future Work

This paper considered the problem of improving the availability of IoT-to-Edge-to-Cloud applications, mainly focusing on cloud aspects. In particular, it proposed a modern cloud deployment approach, leveraging WordPress as a relevant example. However, the proposed approach is wider than WordPress and could be applied to any other large web application to build a robust, high-available, and cost-effective system.

The architecture proposed was implemented to test its effectiveness and released as a set of architecture-ready-to-use templates which represent a starting point for companies or developers with limited prior experience to understand how to effectively deploy and maintain a large web application in a cloud environment. Evaluation results show that selecting

Step number	Template name	Boot time	Description
1	vpc	128 s	Create the virtual private cloud environment
2	sgs	27 s	Setup network security groups and firewall rules
3	cache	251 s	Launch elastic-cache computing nodes
3	alb	72 s	Setup load balancer with specified rules
3	bastion	150 s	Launch SSH accessible node for maintenance
3	rds	589 s	Instantiate redundant managed DB nodes
3	efs	106 s	Instantiate a network block file-system
4	cloudfront	80 s	Setup AWS CDN rules
4	web	281 s	Launch computing nodes and setup the application
5	route53	39 s	Setup DNS rules
6	dashboard	8 s	Create a monitoring dashboard for maintenance

TABLE I
BOOTSTRAP TIME (S) REQUIRED FOR EACH TEMPLATE.

an appropriate scaling policy/metric is fundamental to keeping the system responsive and resilient to unexpected traffic peaks, especially when dealing with a system in which high availability is crucial.

In the future, we plan to enrich templates with a ready-made staging environment to test updates and changes in a non-production environment before proceeding to the actual release. Furthermore, we are actively working on measuring the real cost-effectiveness with respect to deploying separate instances.

Concerning more general work about availability in the IoT-Edge-Cloud, there is plenty of space for future research. First, future research will explore the availability of the IoT/Edge. One relevant challenge to be tackled consists in always guaranteeing a reliable network connection to devices. This could be possibly achieved by *redundancy*, e.g., by configuring IoT devices with multiple alternatives for connectivity.

Monitoring is another key means to achieve availability: IoT, edge, and cloud devices must be constantly monitored to promptly react to anomalies such as faults or security attacks.

Most important, research on *security* in virtualized environments [22], [23], [24] is a key direction to guarantee availability. This is because, for example, attackers can harm the availability of a web service by flooding it with millions of requests, thus triggering a Distributed Denial-of-Service (DDoS) attack. This allows, for example, preventing malicious users from flooding the web service with millions of requests, thus harming its availability or severely compromising performance, thus inevitably impeding to meet Service Level Agreement (SLA) guarantees.

ACKNOWLEDGMENT

This work has been partially supported by the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456.

ONLINE MATERIAL

The documentation, templates, and scripts repository can be found at:
https://gitlab.retis.santannapisa.it/ga.serra/wpaas-architecture

REFERENCES

[1] M. Amin-Naseri, P. Chakraborty, A. Sharma, S. B. Gilbert, and M. Hong, "Evaluating the reliability, coverage, and added value of crowdsourced traffic incident reports from waze," *Transportation Research Record*, vol. 2672, no. 43, pp. 34–43, 2018.

[2] Baidu, "Apollo autonomous driving framework."

[3] L. Belluardo, A. Stevanato, D. Casini, G. Cicero, A. Biondi, and G. Buttazzo, "A multi-domain software architecture for safe and secure autonomous driving," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 73–82, 2021.

[4] P. Zhang, M. Zhang, and J. Liu, "Real-time hd map change detection for crowdsourcing update based on mid-to-high-end sensors," *Sensors*, vol. 21, no. 7, 2021.

[5] M. Little and M. Mullenweg, "Democratize publishing: The freedom to build. the freedom to change. the freedom to share.," tech. rep., Wordpress.org, 2003.

[6] Convesio, "Scalable, high-speed infrastructure for wordpress."

[7] S. WP, "Can wordpress scale?."

[8] BigCommerce, "Is your wordpress website scalable enough for more traffic?."

[9] J. Farmer, "How to scale wordpress to half a million blogs and 8,000,000 page views a month," tech. rep., wpmudev, 2013.

[10] P. Lewis, R. Guilfoyle, A. Chatzakis, and J. Touzi, "Best practices for wordpress on aws," tech. rep., AWS, 2019.

[11] L. McDaniel, "Performance tuning aws efs for wordpress."

[12] J. Geerling, "Getting the best performance out of amazon efs."

[13] K. Konstanteli, T. Varvarigou, and T. Cucinotta, "Probabilistic admission control for elastic cloud computing," in *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 1–4, 2011.

[14] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[15] H. Yang and Y. Kim, "Design and implementation of high-availability architecture for iot-cloud services," *Sensors*, vol. 19, 2019.

[16] A. Kanso and Y. Lemieux, "Achieving high availability at the application level in the cloud," in *2013 IEEE Sixth International Conference on Cloud Computing*, pp. 778–785, IEEE, 2013.

[17] D. Casini, A. Biondi, and G. Buttazzo, "Task splitting and load balancing of dynamic real-time workloads for semi-partitioned edf," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2168–2181, 2021.

[18] IBM, "Three-tier architecture."

[19] A. AWS, "Target tracking scaling policies for EC2 Auto Scaling."

[20] A. AWS, "Configure instance weighting for Amazon EC2 Auto Scaling."

[21] A. AWS, "AWS cloudwatch alarms."

[22] O. Osanaiye, K.-K. R. Choo, and M. Dlodlo, "Distributed denial of service (ddos) resilience in cloud: Review and conceptual cloud ddos mitigation framework," *Journal of Network and Computer Applications*, vol. 67, pp. 147–165, 2016.

[23] N. Borgioli, M. Zini, D. Casini, G. Cicero, A. Biondi, and G. Buttazzo, "An i/o virtualization framework with i/o-related memory contention control for real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, 2022.

[24] A. B. de Neira, B. Kantarci, and M. Nogueira, "Distributed denial of service attack prediction: Challenges, open issues and opportunities," *Computer Networks*, vol. 222, p. 109553, 2023.