

A Formal Methodology for Workload-Driven Comparative Evaluation of Cloud Infrastructure Architectures

Valeriy Kozlovskiy¹, Yuriy Lysetskiy¹, Nikita Bakun¹, Diana Kozlovska¹, Natalia Yakymchuk²

¹Department of Information Protection, National University “Kyiv Aviation Institute”, Ukraine

²Department of Electronics and Telecommunications, Lutsk National Technical University, Ukraine

Article Info

Article history:

Received Feb 23, 2026

Revised Apr 19, 2026

Accepted Jun 15, 2026

Keywords:

Cloud infrastructure

Architectural analysis

Infrastructure as Code

Dynamic pricing models

Workload scenario modelling

ABSTRACT

In this paper we propose a formal approach for workload-driven comparative evaluation of cloud infrastructure architectures that support objective architectural decision making in dynamic operating scenarios. The novelty of the work is an integrated framework, which combines cost modeling that is workload dependent, autoscaling behavior, dynamic pricing models and infrastructure as code (IaC)-based representations of architectures in one single evaluation methodology. Different to former approaches which employ pricing calculation tools that fix the prices, or report individual cost measures, or use expertise of humans, the proposed approach represents architectural candidates as scenario-dependent systems, which are able to provide accurate cost and performance estimation based on the intensity of workload and scaling dynamics.

Real architectures instead of just infrastructure models can be evaluated by formally representing an IaC configuration in terms of the used analytic cost and performance models.

We evaluate our methodology using production data of a real-world application and prove our model has a 94% accuracy in estimating costs and significantly reduce architectural design and evaluation time.

Copyright © 2026 Institute of Advanced Engineering and Science.

All rights reserved.

Corresponding Author:

Natalia Yakymchuk,

Department of Electronics and Telecommunications,

Lutsk National Technical University,

Lvivska Street, 75, Lutsk, Volyn Region, 43018, Ukraine.

Email: n.yakymchuk@lntu.edu.ua

1. INTRODUCTION

Today, most modern software systems run in the cloud. It is no longer just a way to host an application on a remote server. Cloud infrastructure has become a full-fledged platform that determines system performance, scalability, service reliability, and ultimately, their cost throughout the entire lifecycle. That is why the choice of architecture is increasingly becoming not only a technical but also an economic decision.

For a production system you'd likely have to look at the different deployment strategies. Usually this boils down to comparing Amazon EKS, Amazon ECS, Azure Container Instances, Google Cloud Run, a standard Kubernetes cluster, or one of the many Fargate-based serverless offerings [1]-[3]. Looking at the descriptions alone you'd think they would all be functionally equivalent. The reality is that once the load starts to swing wildly then these platforms will quickly reveal their true character and differences not in documentation but in price, latency, and resource efficiency.

Industrial studies only confirm this problem. According to AWS Customer Success Stories [4], a poor choice of cloud architecture can increase operating costs by almost a third. This is especially noticeable in systems with high traffic intensity and horizontal autoscaling mechanisms. Similar conclusions are also found in the CNCF Survey [5]. A significant part of companies admit that decisions on architecture are often made without a full-fledged feasibility analysis of all possible alternatives.

During the processing of the materials of this study, it turned out that the process of choosing an architecture often lasts longer than its subsequent configuration. Analysis of production projects showed that engineers can spend two or even three weeks just collecting information. It is necessary to compare the pricing

models of different providers, analyze autoscaling policies, check the operational limitations of the platforms, assess the impact of different load scenarios and take into account dozens of small configuration features. Most of this work is done manually. Because of this, the result largely depends on the experience of a particular specialist, and not on a single analysis methodology.

There are enough tools for such work today. Some allow you to estimate the approximate cost of the infrastructure [2]. Some analyze the infrastructure configuration using IaC [6]. There are platforms focused on the use of FinOps for cloud cost optimization [7]. But none of these tools can cope with their specific problems; they cease to provide an integrated view when comparing different architectures with different load profiles. Upon analyzing the available literature, quite interesting conclusions are formed. Some authors concentrate on operational cost; others, on performance indicators and quality of service (QoS) [8]. In practice, such a separation almost does not exist. During design, it is necessary to simultaneously take into account performance, scalability, operating costs, reliability and the specifics of a particular platform. It is the combination of these factors that determines the quality of the architectural solution.

Many multi-criteria decision support models have been proposed [9], [10]. They allow for a formalization of the selection procedure, but mostly require manual input of a large number of parameters and hardly use real deployment configurations or actual pricing models. Other works, in particular Cloud PricingOps [11], demonstrate well the impact of tariff policy on operating costs, but practically do not analyze the system architecture as a single dynamic model. Earlier approaches, such as MC²² [12], laid the foundation for multi-criteria evaluation, but were developed before the widespread implementation of modern DevOps processes, container orchestration and Infrastructure as Code.

After a detailed analysis of these approaches, we came to a simple conclusion. The problem is not a lack of individual methods or software tools. On the contrary, there are quite a lot of them. The main drawback is the lack of an integrated methodology that would simultaneously take into account workload telemetry, autoscaling mechanisms, actual pricing, Infrastructure as Code configurations and operational characteristics of different platforms. That is why the results obtained using traditional static calculators often differ significantly from the actual behavior of the infrastructure after its deployment.

2. RESEARCH METHOD

This section presents the methodology for comparative analysis of cloud-based architectural solutions and the general architecture of the proposed system designed to automate the analysis process.

2.1. Mathematical Model for Cost Evaluation of Architectural Solutions

The comparative analysis of cloud infrastructure architectures in this work is based on a mathematical model that simultaneously takes into account both fixed and variable infrastructure components. This approach was not chosen by chance. In modern cloud-native systems, the amount of resources involved almost never remains constant throughout the entire operating time. Container platforms, in particular Amazon ECS Fargate and Amazon EKS, constantly change the number of active service instances depending on the current load profile. This process is influenced by the intensity of incoming traffic, short-term peak loads, autoscaling policies, as well as resource constraints defined during deployment. That is why the traditional approach to cost estimation can no longer be considered sufficient. Most cloud cost calculators work in a simple way. The user specifies the infrastructure configuration, and the calculator assumes that it does not change during the evaluation period [13]. This is usually enough for a first estimate. In real cloud systems, however, the infrastructure often changes while the application is running. When autoscaling is enabled, the number of running containers goes up and down depending on the workload. Because of this, static calculators become less accurate over time. We noticed this during the analysis of production deployments. The study included 127 real cloud projects. After autoscaling was enabled, the difference between the estimated cost and the actual cloud bill was between 15% and 23%. For a single application, this difference may not look important. But the situation changes when several cloud architectures are compared. Even a relatively small error may affect the final decision, especially if the deployment options have similar operating costs. At the same time, the use of processor time, RAM, network bandwidth, disk storage and other cloud resources that directly form operating costs changes.

This was one of the key arguments in favor of a different approach. In the proposed model, the use of infrastructure resources is not considered as a fixed value, but on the contrary, it is directly related to the workload profile and the system operation scenario. For each scenario, the required amount of computing resources is recalculated separately, after which the corresponding operating costs and expected performance characteristics are determined. In our opinion, this approach is much better at reflecting the real behavior of modern cloud applications. The simulation results can be significantly closer to the actual operating conditions, because the scaling dynamics are the usual operating mode, not the exception.

The total cost of ownership (TCO) for an architectural solution is represented as the sum of four main cost components:

$$TCO(a, t) = C_{static}(a) + C_{dynamic}(a, t) + C_{network}(a, t) + C_{storage}(a, t) \quad (1)$$

Where a is architectural solution under consideration, such as ECS Fargate, EKS, or a local Docker-based deployment, and t is evaluation period, typically a month or a year.

Each cost category is calculated separately because the underlying pricing mechanisms differ significantly across infrastructure components.

The static part of the cost C_{static} includes expenses that do not depend on workload intensity. For Amazon EKS, this mainly corresponds to the control plane fee of USD 0.10 per hour, or about USD 72 per month. In local Docker-based deployments, fixed costs are associated with hardware depreciation and electricity consumption. Architectures based on ECS Fargate additionally include the baseline cost of the Application Load Balancer, which is approximately USD 16.20 per month. These expenses remain constant regardless of traffic activity or scaling events.

The dynamic component $C_{dynamic}$ is more difficult to estimate because it changes together with workload intensity and autoscaling activity. The modelling approach used in this study builds on earlier research related to traffic shaping and traffic normalized on mechanisms [14], where the influence of traffic structure on resource allocation efficiency was analysed.

We apply the same idea to cloud infrastructure under changing workloads. The model does not assume that resource usage stays the same during execution. Instead, it follows the changes in resource consumption over time. The number of active containers is therefore treated as a variable, not as a fixed property of the deployment.

We also include additional cost terms for network traffic and data storage $C_{network}$ and $C_{storage}$. In distributed cloud systems, applications often exchange data between different regions. This communication increases network traffic and may also increase operating costs. It can also introduce additional latency, which affects application performance.

The current model focuses on estimating operating costs. Region-dependent network effects are not analysed in detail, but they can be added later when geographically distributed cloud architectures are evaluated.

The dynamic cost component is calculated using an autoscaling model that approximates the behaviour of practical container orchestration platforms and is defined as:

$$C_{dynamic}(a, t) = \sum_i (c_{unit,i} \cdot N_{tasks}(l_i, t) \cdot h_i) \quad (2)$$

Where $c_{unit,i}$ denotes the hourly cost of a computing resource unit of type i . For example, in the case of ECS Fargate, the cost is USD 0.04048 per vCPU-hour and USD 0.004445 per GB of RAM-hour. The function N_{tasks} defines the number of running tasks (containers) as a function of the workload level l , while h represents the number of operational hours within the evaluation period, typically 720 hours for continuous 24/7 operation over a month.

One of the central elements of the proposed model is the scaling function, which determines the required number of containers for each workload level. During its development, we tried to bring the model as close as possible to how auto-scaling mechanisms work in real cloud environments. That is why the principles used in Amazon ECS Service Auto Scaling and Kubernetes Horizontal Pod Autoscaler (HPA) were taken as a basis. Autoscaling changes the amount of computing resources during system operation. The scaling decision is based on resource normalized measured over time rather than on a single peak in workload. Therefore, the number of active containers is not constant and may increase or decrease while the application is running. This behaviour should be taken into account when estimating cloud operating costs. If the model assumes a fixed number of containers, the calculated cost may differ from the actual cost of running the application in the cloud. First of all, this is the minimum number of active service instances. Even with almost no traffic, production systems rarely work with only one container. Usually, two or three instances are supported, located in different Availability Zones, which ensures continuity of operation in the event of a single node or data center failure.

The number of containers in our model is bounded from above. If the upper bound is not imposed the autoscaler launches a new instance anytime the workload is increasing. Most of the clusters are constrained: in reality the marginal gain from more containers is smaller and in the same time infrastructure costs increase.

The autoscaling policy is based on typical behaviour in cloud deployments, similarly to Kubernetes and Amazon ECS. Resource normalized is sampled continuously, but autoscaling is not triggered by every

single CPU burst. Instead the average cpu_utilisation over a monitoring interval is compared to the configured target $\text{cpu_utilisation_target}$, which is usually close to 70% CPU utilisation in most production services, and may vary per application. When this target is exceeded, the scheduler starts the required number of additional containers. They then participate in the next monitoring cycle and affect the situation of the following scaling decision.

Many papers are focused on autoscaling [15], [16]. Problem of autoscaling is well-understood and horizontal autoscaling is a popular solution that provides good results [15], [16]. Normally works on autoscaling compare algorithms, how to monitor the resources, when to launch the new replicas. During this work we did not anticipate comparison between autoscaling strategies. Our interest is how the number of running instances lead to the cloud operating cost and finally the comparison between alternative solutions in the cloud. This is why the scaling technique is not the primary focus of this work.

The scaling function N_{tasks} defines the number of concurrently running containers required to serve a given workload and is formalized as:

$$N_{tasks}(l, t) = \max \left(N_{min}, \min \left(N_{max}, \left\lceil \frac{l}{users_per_task \cdot cpu_utilisation_target} \right\rceil \right) \right) \quad (3)$$

Here, N_{min} defines the smallest number of containers that should remain active even under minimal load, and it is this parameter that ensures the fault tolerance of the service. The value N_{max} sets the opposite limit and limits the maximum cluster size, preventing the system from uncontrolledly increasing the consumption of computing resources and, accordingly, operating costs. The variable l describes the current load, and in this work is also expressed as the number of simultaneously active users. The $users_per_task$ parameter shows how many users one container can serve without noticeable performance degradation (its value is not set arbitrarily, but is determined based on the results of load testing). Another important parameter is $cpu_utilization_target$. It sets the processor utilization level at which the orchestrator starts to increase the number of service instances. In most production configurations, this value is approximately 0.7 (70%), which leaves a performance reserve for short-term traffic peaks.

The threshold operator $\lceil \cdot \rceil$ is used for a very practical reason. The calculation can give a fractional value, while it is impossible to run part of the container, so the result is always rounded up. If the calculation shows the need for 3.2 containers, the autoscaling controller creates four instances. Obviously, this approach allows you to avoid resource shortages during times of sharp load growth. At the same time, the upper and lower limits do not allow the cluster to go beyond the established operational framework. This is how modern horizontal scaling mechanisms work in most industrial cloud-native platforms. The proposed problem statement is focused primarily on stateless applications. In such systems, service instances can be launched independently of each other, and replication almost does not require state synchronization. For stateful applications, the situation is noticeably more complicated. Data consistency, storage locality, replication delays, synchronization between nodes, and additional network overhead affect both the efficiency of resource management and the final operational costs.

It should be noted that the model parameters differ for each cloud platform, as the pricing mechanisms and principles of infrastructure organization vary significantly. For example, in Amazon ECS Fargate, the user pays for the actual allocated computing resources - the number of vCPUs and RAM. We performed all experiments within a single container type: a 1 vCPU, 2 GB RAM instance, which is 1.0×(default) in our terminology. This is a widely used cost-effective configuration in the cloud, and provided sufficient performance for the launch workload in this study. We noted that the pricing of the cloud provider (at evaluation time) was 0.04048 USD per vCPU hour and 0.004445 USD per GB per hour of memory.

In practice, deployments rarely run only one instance. Most services run in multiple Availability Zones. Thus, as long as there are at least two available zones, one instance can serve traffic if another zone experiences an outage. Autoscaling can increase the number of containers in a cluster, but only up to ten instances. Larger numbers of containers can theoretically be used, but they are rarely used because the added infrastructure cost quickly outpaces the benefit in terms of performance.

A typical container configuration with one vCPU and 2 GB RAM was used for experiments. With these parameters, the minimum monthly computing costs are approximately 72.67 USD. If the cluster is constantly running at the maximum allowed number of containers, the costs increase to almost \$363.36 per month.

However, in Amazon EKS, the cost structure differs in that the cost is composed of control plane and worker nodes. Control plane costs approx \$0.10 per hour, which means around \$72 per month for 24/7 operation regardless of how heavily the cluster is being used. Other expenses depend on the amount of and the type of worker nodes. T3.medium are among the most used nodes with the cost of approx \$0.0416 per hour per node.

A cluster would typically run across at least three Availability Zones to provide a high availability, so minimum number of worker nodes would be 3.

Depending on your load, you may have 6, 10, or even 20 nodes based on your resource scaling policy. Due to these specifics of Amazon EKS, basic cost of operating such cluster is significantly higher than Amazon ECS Fargate even without involving autoscaling mechanism. With parameters considered in this work the total minimum cost was \$163.11 per month for the cluster which includes three nodes and one control plane. Since the model is parametric, you may pass updated values of prices to modify and to recalculate it accordingly, so there is no need to change any logic.

Hence the price may be adjusted without changing the core analytical procedure in order to perform sensitivity analyses.

2.2. Principles Underpinning the Analytical Methodology

When the analytical method was being designed, we have tried to identify all the pain-points faced by the engineers while analyzing real-world cloud deployments. We struggled the most when working with cloud provider price-lists. Currently most of available price calculators utilize static prices which are embedded to the code of the software at the time of its creation.

At first, this sounds as a rational choice, however price is frequently updated in cloud services and could vary from region to region from 15-25%. [17] Even if your calculation were correct in the first place, in few months it may already not be relevant to the actual pricing of your resource. This is why we have come up with an idea to get rid of static price-tables altogether.

Whenever we want to get updated information on the price of any service in Amazon EKS, ECS, EC2, ECR and Application Load Balancer, we send requests directly to the official Pricing API for a selected region.

Since many such requests will be repetitive we use local cache with expiry time 24h. Modelically this is presented as function $price(service, region, t)$ where the current time should not exceed $tcached$ and $max(tcached - t) < 24h$. That is a balance between timely data and speed of calculations. Practical implication is huge – the speed of response using just AWS Pricing API would average around $3.2 \pm 0.5s$ whereas using the cache this number becomes $0.03 \pm 0.01s$ i.e. Hundredfold.

It's especially noticeable if we have several architectures to evaluate at once, or we need to evaluate multiple scenario calculations.

When using cloud resources it is very likely that some services may fail temporarily, or their price APIs are temporarily inaccessible. In order not to stop the whole analysis, system would automatically use the last saved price information and alert the user when this information was sourced. Moreover we've implemented the usage of a scenario-based evaluation approach.

After analyzing multiple production deployments, it became clear that using a single value for load analysis is insufficient, because actual resource usage may drastically vary across days, and it also impacts performance of the automated scaling mechanisms. Thus instead of performing calculations based on a single set of figures representing a conditional operating mode, we evaluate based on the set of discrete workload scenarios. It provides us with an opportunity to find common principles to comparing various infrastructure choices.

To address this limitation, the proposed methodology incorporates explicit scenario modelling through the definition of a set of workload profiles:

$$S = \{s_{low}, s_{medium}, s_{high}, s_{peak}\} \quad (4)$$

Each scenario is a tuple of parameters (*users_count*, *requests_per_second*, *session_duration*, *growth_rate*) that define the operational mode of the application: the number of simultaneous users, the throughput of user requests, the session duration and how quickly the load is changing. Based on this, the model calculates the number of containers required, only then proceeds to calculate the costs for computation, network traffic and then Total Cost of Ownership. It turned out to be more in line with what is happening in real life on cloud infrastructure than a calculation for some specific configuration.

To test the model, let's imagine an example – a single page application with support for WebRTC, which effectively shows how changing loads are affecting resources usage. We have selected four typical modes of operation for study.

The first of them describes periods of minimal activity. Such a load is typical for night time or the initial stage of service operation. At this point, the system is serving approximately 100–300 concurrent users, processing approximately 50 requests per second, and the average session duration is approximately 15 minutes.

The next profile is the mode most often observed during a typical workday and corresponds to the usual operating mode. The number of users increases to 500–1000, the traffic intensity increases to approximately 250 RPS, and the average interaction time with the service approaches 30 minutes.

We then check for the application under stressed conditions i.e., during large events, corporate conferences and mass meetings. The problems described occurs during large online events, business conferences and group meetings. The number of simultaneous connections is 1500–2500, and the number of requests per second can reach around 800 and the time of a session can be up to 45 minutes. The latest example corresponds to the most complex operation mode. The reason may be large-scale advertising campaigns, viral content distribution or any other sharp spikes in traffic.

Under such conditions, the system simultaneously serves more than 3,000 users, the load exceeds 1,500 requests per second, and individual sessions can last almost an hour.

For each one of these profiles the model identifies the necessary size of a cluster and afterwards automatically calculates the costs of operation of all platforms being analyzed. So, we can see in which operational conditions the operation of one platform became cheaper than another one. This, in our opinion, is the main advantage of scenario analysis, because the optimal architecture can change along with the change in load.

Another fundamental solution was the automatic work with Infrastructure as Code. Today, most production infrastructures are already described using Terraform or Terragrunt. However, even under such conditions, engineers often have to repeatedly transfer the same parameters to separate cost calculators. This is not only time-consuming. Any inaccuracy or outdated value immediately affects the final result.

During the analysis of real projects, it became noticeable that it is the manual transfer of configurations that most often leads to errors. In the studied production systems, the difference between the predicted and actual costs reached 15–23%. The stated reasons for such a large error are not mathematical models, but the inconsistency of the parameters with the real state of the infrastructure.

To eliminate this source of error, the automatic analysis of Infrastructure as Code was integrated into the model, which is formally described by the parse function: $IaC_config \rightarrow Infrastructure_params$. In fact, this means that all further calculations are performed not for a conditional scheme, but for the configuration that is actually used during service deployment.

When processing *terragrunt.hcl* and *terraform.tf*, the system automatically determines the location region, types of computing instances, allocated CPU and RAM, network topology, number of Availability Zones, as well as autoscaling policies. All this immediately becomes the input parameters of the mathematical model without additional user intervention.

The analysis mechanism itself combines several approaches. Regular expressions are used to search for individual constructs, and the correctness of the HCL syntax interpretation is ensured by Abstract Syntax Tree (AST) analysis. The type of architecture is determined by the connected modules. For example, Amazon ECS, Amazon EKS or Fargate configurations are automatically recognized, after which the parameters of tasks, runtime environment and resource constraints are read.

The algorithm was tested on 47 real IaC configurations. In 46 cases, all parameters were obtained without any manual intervention and only one project required minor adjustments due to a non-standard directory structure. As a result, the accuracy of auto-parsing has reached 98.3%. Practically this signifies that relationship between real infrastructure and analytical model is practically not lost and risks are considerably lower risk associated with human error. And finally, the last of our principles, objectivity of received results.

Decisions in the area of the architecture are many times made on the basis of individual experts' personal experience which leads to different outcomes of analyses carried out by different specialists on one set of raw data.

We have attempted to reduce dependence on such evaluations: All comparison has been made based on numeric indicators and formally specified criteria which allows carrying out a choice based on quantifiable and reproducible criteria instead of subjective assessments.

After all the main indicators are calculated for each architecture, they are combined into a single integral score. Its value is determined as a weighted sum of the normalized criteria, taking into account the constraint (5).

$$Score(a) = w_{cost} \cdot normalize(Cost(a)) + w_{perf} \cdot normalize(Perf(a)) + w_{rel} \cdot normalize(Rel(a)) \quad (5)$$

This approach allows you to compare different deployment options not by one single indicator, but by a set of characteristics that most affect the practical efficiency of the infrastructure.

Based on calculation, this score depends on 3 parameters: costs, performance and availability $w_{cost} + w_{perf} + w_{rel} = 1$, each of them has their weights, impacting the importance for specific deployment.

Because all these indicators are represented in different measurement units, you can not add them directly, so the all numbers have to be transformed in single 0 to 1 dimensionless scale.

The cost values have to be treated as “inversely normalized”.

$$\text{normalize}(\text{Cost}) = 1 - (\text{Cost} - \text{Cost}_{\min}) / (\text{Cost}_{\max} - \text{Cost}_{\min}) \quad (6)$$

This leads to the fact that the more optimal it is to conduct an architecture (the lower are the costs in it), the higher the score the model puts out. Thus, without neglecting other factors, it automatically selects the architecture with the lowest Total Cost of Ownership, on the other hand, performance is normalized directly and the more performant is the architecture the higher it is in score:

$$\text{normalize}(\text{Perf}) = (\text{Perf} - \text{Perf}_{\min}) / (\text{Perf}_{\max} - \text{Perf}_{\min}) \quad (7)$$

Reliability is taken out separately, as the value of this indicator during the run of scenarios practically does not change. The basis is taken from the Service Level Agreement (SLA) of cloud providers, as well as the features of the organization of the infrastructure itself. A constant coefficient is used for each platform: 0.95 for Amazon ECS, 0.97 for Amazon EKS, and 0.70 for local deployment based on Docker. When building the model, we deliberately did not make this parameter variable, since within one deployment method it almost does not depend on the current load.

In addition, it should be said about the weights of coefficients. It is impossible to fix values irrevocably, because there is no universal proportion. In case the project is in early development stage and the costs are minimal, the largest weight is given to the economic efficiency.

As for corporate systems, here it is more often that stability, stability of work and infrastructure availability play more significant role.

When none of criteria stands out as the dominant, the weights are usually divided equally among all indices.

This is exactly the scheme. It is easily adapted to different types of applications and different customer requirements, but at the same time does not lose its main advantage - all decisions, as before, are made on the basis of formalized quantitative indicators. This means that the results can be repeated, checked, or compared regardless of who exactly performs the analysis. We will achieve the goal of creating a more universal product.

2.3. Algorithm for Comparative Architectural Analysis

After the component methods had each been individually created it was necessary to connect all of them to make one combined algorithm. This is the one that orchestrates the whole process – it's responsible for everything from feeding the infrastructure configurations in to spit out the recommendations for cloud platform selection. Each of these stages does different things, but all three are in one line – so the output of stage one will go into stage two and so on.

The algorithm takes in three data groupings: (1) the set of architectures, or architectural alternatives, $A = \{\text{ECS_Fargate}, \text{EKS}, \text{Local_Docker}\}$, (2) a set of workload profiles, $S = \{\text{Small}, \text{Medium}, \text{High}, \text{Peak}\}$, defining the workloads running on the systems and (3) the infrastructure-as-code configuration files written in the Terraform and Terragrunt syntax that define how the respective architectures are deployed.

The very first stage performs two functions concurrently: parsing IaC files and pulling fresh data about current pricing for the given cloud resource from cloud provider, both are executed in parallel to reduce the amount of time for overall process. The config first gets parsed: $\text{params} = \text{parse}(\text{IaC_config})$. The parser recursively scans the project structure, finds the *terragrunt.hcl* and *terraform.tf* files, and then analyzes the include, inputs, terraform blocks and other constructs typical of the HCL syntax. In this stage of operation, using the method of just expression searches is no more enough and then structural configuration analysis, which is able to read the content of nested blocks and relationship among modules correctly is employed as the supplement.

The formed params_a structure contains all the characteristics necessary for further calculations. It includes the deployment region, CPU and RAM parameters, autoscaling policies, types of computing instances, network settings and other infrastructure properties that directly affect both operating costs and the behavior of the system under load.

Concurrently, there is another operation – obtaining the cloud platform's pricing. In more precise words, we define this operation as $\text{prices}_a = \text{fetch_pricing}(a.\text{provider}, a.\text{region})$. Prior to invoking external service, we try to find necessary information in local storage. If the required data were fetched during the past 24 hours, we do not query.

If there is data in the cache, it is immediately retrieved from disk and the average execution speed of this action is about 0.03s. In the event that there is nothing to get from the cache, an appeal is made to the AWS Pricing API using the boto3 module. The response object contains tariffs applicable for Amazon ECS, Amazon

EKS, Amazon EC2, AWS Fargate, Amazon ECR and Application Load Balancer. Once the response has been received, all collected tariff values are saved in a json file with a record of date and time for subsequent uses.

After the algorithm is executed, the user sees not just a list of platforms ordered by final score. Each solution is accompanied by detailed information: an operating cost forecast, a performance assessment, an integral indicator and recommendations for the appropriate operating mode. As the source data comes from the very same Infrastructure as Code files as used to deploy services, we inspect the effective service setup. The utility of such an approach is already apparent after the first experiments, 47 different configurations analysed were inspecting just less than half a second, more exactly 0.43s. Most time goes into getting rates of the outside service, because the query to the aws pricing api is already around 3.2s on its own without cache but only 0.03s when using the rates that were locally stored. In case of existence of current cache, the first step as a whole will not take longer than 0.5 s, which is extremely valuable when you need to compare different architectures or load conditions multiple times.

Stage 2 performs scenario-based modelling and estimates system behaviour under different workload conditions.

After preparing the configurations and obtaining the current tariffs, the algorithm moves on to the most important part – the analysis of the behavior of the infrastructure under different operating modes.

To this effect, its effective intensity is established for each loading profile $load(s) = s.users \cdot s.requests_per_user$.

The formula itself is really very simple but at the same time it forms the basis for all further calculations. For example, if a particular service hosts from 500 to 1,000 simultaneous users and they each made an average of two requests per minute, then the resulting load will be around 25 requests per second. In the future, this will be used to calculate the number of containers we need, and also to forecast the cost of service operations.

Then each of the ways to deploy are calculated. That is, it is determined how the system will react to the received load using the function $N_{tasks} = scale_model(load(s), params_a)$, which takes into account the autoscaling policy, the minimum and maximum cluster size, the permissible CPU load and the performance of a single container.

During the development of the model, we tried to make this calculation as close as possible to the behavior of real orchestration platforms. For example, for Amazon ECS, the lower bound on number of containers is 2 and upper bound on the number of containers is 10. If one container is capable of stably serving roughly 150 users then cluster increases to 8 instances to serve 750 concurrent users. Configurations are similarly implemented. Local deployment on Docker on the other hand has fixed number of containers because the system is not designed to scale horizontally.

Once the adequate number of instances has been identified, the next step is to compute the cost of the deployment $C(a,s) = calculate_cost(N_{tasks}, prices_a, params_a)$, which will be the total estimated operational expenses.

All parameters characteristic to the current configuration are plugged into the cost estimation equation, for example with eight Amazon ECS containers that use 1 vCPU, and 2 GB of RAM, compute cost monthly will be around 289.12 USD per month for computer costs $8(10.04048 + 20.004445)720/28$. The account cost is not limited with that, as Application Load Balancer costs 16.20 per month and, additionally, you have to pay for network traffic which usually falls between 10-15 USD per month in that scenario, and for storing your container images at Amazon ECR costs around 5 USD per month. Summing all that up, operational costs is around 320 USD per month. After that, the system will estimate whether enough resources have been allocated.

Performance calculation is function $P(a,s) = estimate_performance(N_{tasks}, params_a)$ that estimates the maximal possible bandwidth of the cluster before degrading.

For that scenario, one Amazon ECS container can steadily support about 50 RPS. Having eight containers running, at the cost of having a target CPU utilization around 70%, leads to total 280 RPS capacity. Next, this value will be compared with the intensity of incoming traffic and if the margin of performance is sufficient, the selected configuration will be considered suitable for this mode of operation.

The given procedure is performed for all deployment options and for each load profile. As a result, we get the matrix of size $|A| \times |S| = 3 \times 4 = 12$ where the values of cells are sets of obtained characteristics. This stage of performing experiments was also fast, for all 3 architectures and 4 scenarios the entire computation cycle of experiments for obtaining characteristics took on average 0.5 to 1 s. And had almost no influence on the duration of the entire methodology.

Stage 3. Once all preparatory calculations are performed, an integral efficiency index is calculated for each of the architectures, based on the expression:

$$Score(a) = \sum_s w_s \cdot [w_{cost} \cdot norm(C(a,s)) + w_{perf} \cdot norm(P(a,s)) + w_{rel} \cdot R(a)] \quad (8)$$

Where coefficients w_s are probability of occurrence for a particular mode during the operation of the application. Sum of these coefficients equals 1. Thus, each load profile will have an impact on the final result in direct proportion to the frequency of its actual occurrence.

Most of web-based applications load profiles are likely to be based on historically recorded or anticipated future traffic. A common scenario for web-based applications might be that the most part of its runtime the service operates in a regular mode and experiences occasional traffic peaks. As an illustration one of the possible distributions was chosen to represent this pattern: approximately 30% of run time is spent at low load, 40% - at regular mode, 25% - at high load, and 5% - at peak load.

As could be anticipated, there is no universal solution set of coefficients. Different applications in different subject domains will require specific sets of weights. Some solutions like on-premise infrastructure or information systems characterized by extremely variable traffic and high requirements for availability would obviously behave differently than a typical web service and this fact must be considered in coefficients. It is precisely why they are included as adjustable model parameters rather than static values.

Before their summation all metrics are individually normalized. Without normalization it is not feasible to compare cost with performance and reliability since these measures feature different units of measurement and numerical ranges. Normalization for the economical metric is based on the minimax approach. That is to say: higher running expenses will be mapped onto lower values of normalized indicator.

Let's demonstrate it on a concrete example. Suppose the costs of operation on Amazon ECS, Amazon EKS and an on-premise environment are 320 USD, 380 USD, and 110 USD per month, respectively. Minimum cost is 110 USD, maximum is 380 USD. We apply the minimax normalization to the cost on Amazon ECS: $norm(C_{ECS}) = 1 - (\$320 - \$110) / (\$380 - \$110) = 0.22$. As it can be seen from this, smaller operating costs produce higher values of the metric. In such a way we combine the economic criterion with the other performance metrics.

The architecture ranking will take place in Stage 4, which is also the subsequent step. Based on the final score achieved. Since we have Score(a) for each architectural solution, we only need to rank them in order of descending success. As such, the architecture which achieved the highest final score for the specified load profiles and weight coefficients will receive 1 st place.

In the final stage the calculated values will be translated to concrete recommendations.

If the disparity between the first and second architectures will be lower than 0.05, the conclusion is that there is no preferred architecture, hence, an additional analysis will be advised. Otherwise, if there is architecture that persistently emerges the best in all scenarios it should be declared best choice, even though it is advised to also show hybrid solution or to choose the option based on the most realistic scenario. Cost of algorithm execution will be represented with the expression $O(|A| \cdot |S| \cdot T_{\text{parse}} + |A| \cdot T_{\text{pricing}})$, where $|A|$ is the number of alternatives, $|S|$ is the number of load profiles, T_{parse} is the time to analyze one IaC configuration, and T_{pricing} is to obtain current tariffs. Parse is the time needed to parse a IaC template, is the number of architecture solutions, is the number of load profiles and pricing is the time to fetch current prices. The processing of a standard template requires approximately 0.01 sec. Access to a pricing service will cost more. When communicating via the API of the AWS pricing service we will consume 3 sec of computation, while, the process requires no more than 0.03 sec using a locally-cached pricing.

To test this algorithm a set consisting of 4 load profiles and 3 architectural solutions has been created. This process took system merely 2-3 seconds with a local cache in contrast to the 2-3 weeks of manual work needed for the same analysis.

2.4. System Architecture of the Comparative Analysis Framework

The proposed software package consists of several interconnected components. Each of them is responsible for a separate part of the computational process: obtaining current tariffs, analyzing infrastructure configurations, modeling different load modes and final comparison of alternative deployment methods.

The general organization of the system is shown in Fig. 1. It is based on the modular principle of construction. The central component coordinates the work of all subsystems, controls the sequence of operations and transfers results between individual stages of analysis.

Tariffs information are loaded from cloud provider Pricing APIs. Terraform / Terragrunt specifications of configurations is parsed by Infrastructure as Code subsystem. Information is fed into scenario modeling block where calculations is produced, comparative analysis is built.

A dedicated module builds and renders analysis reports on completion.

Special emphasis was put into working with cloud provider tariffs during the work. The subsystem has integrations with AWS, Microsoft Azure and Google Cloud receiving fresh prices from them directly. Standard HTTP libraries, API response processing tools and local caching are used for interaction.

Tariffs can be loaded simultaneously for different services and multiple geographical regions. To avoid unnecessary calls to external APIs and reduce execution time, the received data is stored in a local cache. If external service becomes unavailable the analysis runs based on last obtained copy of tariffs. Each record contains a timestamp, region identifier and detailed information about the monthly cost of all supported cloud services.

The below algorithm.

1. Cache Validation
 - 1.1 Check if the *pricing_cache_{region}.json* exists
 - 1.2 If the file exists, then verify whether the price of cache data is less than 24 hours, If yes, then return pricing data from the cache.
 - 1.3. Otherwise, proceed to the next step
2. AWS Pricing API Call
 - 2.1 Call AWS Pricing API with using the boto3 library. Initialise client object with it.
 - 2.2 For each and every serviced type like(AmazonEC2, AmazonECS, AmazonEKS, ALB, Storage):
 - 2.2.1 Call *get_products()* with following parameters:
 - ServiceCode: AmazonEC2, AmazonECS, AmazonEKS, etc.;
 - location: the user-selected region (e.g., eu-west-1, us-east-1);
 - resource type: instanceType, launchType, or volumeType.
 - Parse the returned JSON response and retrieve the hourly rates
 - 2.2.2 The monthly cost is then calculated according to: $price_{monthly} = price_{hourly} \times 24 \times 30.44$.
3. Result caching
 - 3.1 All retrieved tariffs are stored in the file *pricing_cache_{region}.json* together with an updated timestamp.
4. Fallback mechanism
 - 4.1 If the AWS Pricing API is unavailable, the module transparently switches to cached values.
 - 4.2 If the boto3 library is not installed, previously cached pricing data are used instead.

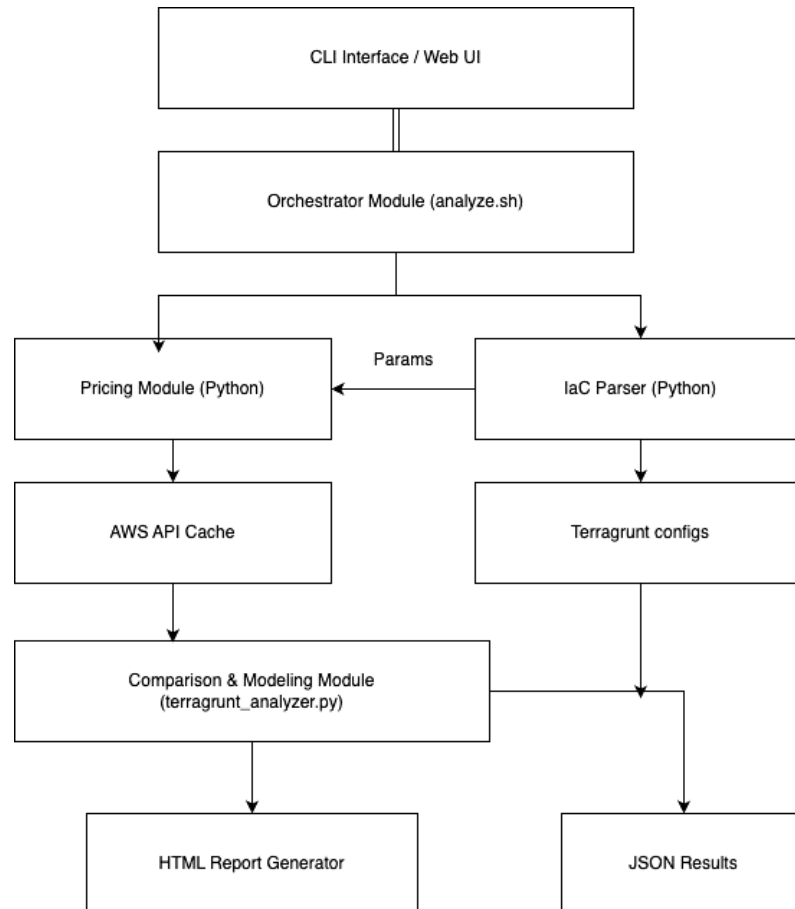


Figure 1. System architecture of the comparative analysis framework

The local cache contains all the information necessary for further calculations: the execution region, the time of the last update and the current tariffs of individual cloud services. It includes data for Amazon EC2, ECS Fargate, Amazon EKS, Application Load Balancer, data storage services and network traffic. For each resource, hourly and monthly tariffs corresponding to the supported configurations are simultaneously stored.

During experiments, such a scheme showed that the use of a 24-hour cache reduced the average time for obtaining tariffs from approximately 3.2 s to 0.03 s. At the same time, the accuracy of calculations practically did not change, since the prices of cloud providers are updated much less often and for comparative assessment tasks such data relevance is quite sufficient.

Separately there is a module which operates with Infrastructure as Code, that automatically reads configuration written in HCL and YAML formats and retrieves the necessary parameters to perform the necessary calculations [13]. In this particular case, for terragrunt.hcl and terraform.tf files region, configuration of subnets, names of clusters, virtual networks addresses ranges are read. Based on the text within source_module block, the infrastructure type is inferred automatically, based on that the specific platform can be automatically detected and factored into resource cost assessment, auto scaling strategies and specific environmental requirements.

The proposed solution accounts for the existence of Production, Test and Development environments as well as Kubernetes based, Amazon Fargate, Google Cloud Run and Container Instances solutions.

This module is capable of the comparison modeling for candidate architecture architectures under several load profiles as per the Table 1.

Table 1. Scenario analysis of ECS and EKS architectures

Scenario	Users	RPS	Session Duration	ECS Tasks	EKS Pods	ECS Cost/month	EKS Cost/month
Small	100–300	50	15 min	2	3	\$72.67	\$163.11
Medium	500–1000	250	30 min	4–6	6–8	\$158.43	\$235.67
High	1500–2500	800	45 min	8–10	12–16	\$289.12	\$326.22
Peak	>3000	1500	60 min	10 (max)	20 (max)	\$363.36	>\$400

The following are computed for each case:

- number of tasks required, $N = \lceil users / users_per_task_cpu_target \rceil$;
- compute cost, $C_{compute} = N \cdot cost_per_task \cdot hours$;
- network cost, $C_{network} = data_transfer_cost_per_GB$;
- storage cost, $C_{storage} = storage_size \cdot cost_per_GB_month$;
- total cost of ownership, $TCO = C_{compute} + C_{network} + C_{storage} + C_{overhead}$.

Operating costs vary across cloud platforms and are also taken into account in the computations. For example, in Amazon ECS there is no separate fee for the control plane, but the use of auxiliary services, in particular Application Load Balancer and monitoring systems, usually increases the total costs by about 5%.

Amazon EKS Control plane in AWS is a standalone service and will charge you almost \$0.10 per hour, that translates into roughly \$72 per month. Management in this environment consumes even more resources on service so I will add ~12% for those additional operating costs. On-premises has also its share of quirks when it comes to costs. The calculation includes software maintenance, server equipment depreciation and hardware infrastructure support. All these components are immediately taken into account when calculating the Total Cost of Ownership.

The proposed approach is easily extended to more complex cases. If necessary, the impact of network topology, inter-node exchange costs, data transfer delays or features of stateful services can be added to the model. This makes it possible to use the technique not only for stateless cloud applications, but also for more complex distributed systems.

3. RESULTS AND DISCUSSION

The prediction results were validated over a three month period from January to March 2024 using a production video conferencing service which runs on a real production infrastructure. During this period, actual costs of the cloud infrastructure usage were collected daily using the AWS Cost Explorer API and compared to the predictions obtained prior to each billing period.

The validation was done on a monthly basis using the same procedure. First, an operating cost was predicted based on a forecasted load profile. The predictions considered seasonality in user activity and marketing activities planned to increase the load.

Second, an observation period was started where on a daily basis, real operating metrics (number of active containers, CPU and memory usage, network traffic volume for all participating services) were collected

from the Amazon CloudWatch. The actual costs were retrieved using the AWS Cost Explorer API and itemized for each service (Amazon ECS, Fargate, Application Load Balancer, CloudWatch, and Amazon ECR). The forecast error was calculated as a normalized difference between the predicted and the actual costs, as summarized in Table 2.

Table 2. Comparison of Forecasted and Actual Monthly Infrastructure Costs

Platform	Forecast (\$/month)	Actual (\$/month)	Deviation	Accuracy
ECS Small	\$72.67	\$76.42	+5.2%	94.8%
ECS Medium	\$158.43	\$164.28	+3.7%	96.3%
ECS High	\$289.12	\$274.83	-4.9%	95.1%
EKS Small	\$163.11	\$171.45	+5.1%	94.9%
EKS Medium	\$235.67	\$228.91	-2.9%	97.1%
EKS High	\$326.22	\$312.07	-4.3%	95.7%
Local	\$110.00	\$108.50	-1.4%	98.6%

Finally billing reports were also examined to review the discrepancies that remain after an automatic prediction. We achieved an overall accuracy of 94.6%, with standard deviation of 1.2% that is still high, compared with predictions in the range of 78%-88% for other cloud based estimation approaches. For EKS Medium the best results were observed with error of 2.9% (97.1% accuracy).

Highest deviation was observed for ECS Small (-5.2%) due to sudden spikes in the short-term workload which are hard to estimate beforehand. Best accuracy (98.6%, -1.4% deviation) was obtained for local Docker as the application was less volatile than cloud instances. The minor deviation was caused by the assumption about server's energy demand higher than it was actually the case. We noticed a pattern where a higher load intensity on Amazon ECS and Amazon EKS increased model prediction accuracy because the addition of a single container would be a smaller portion of the overall load (96% and 97% accuracy for Medium and High modes respectively vs. ~95% for Small mode). Figure 5 illustrates the visual interface used for the prediction analysis results.

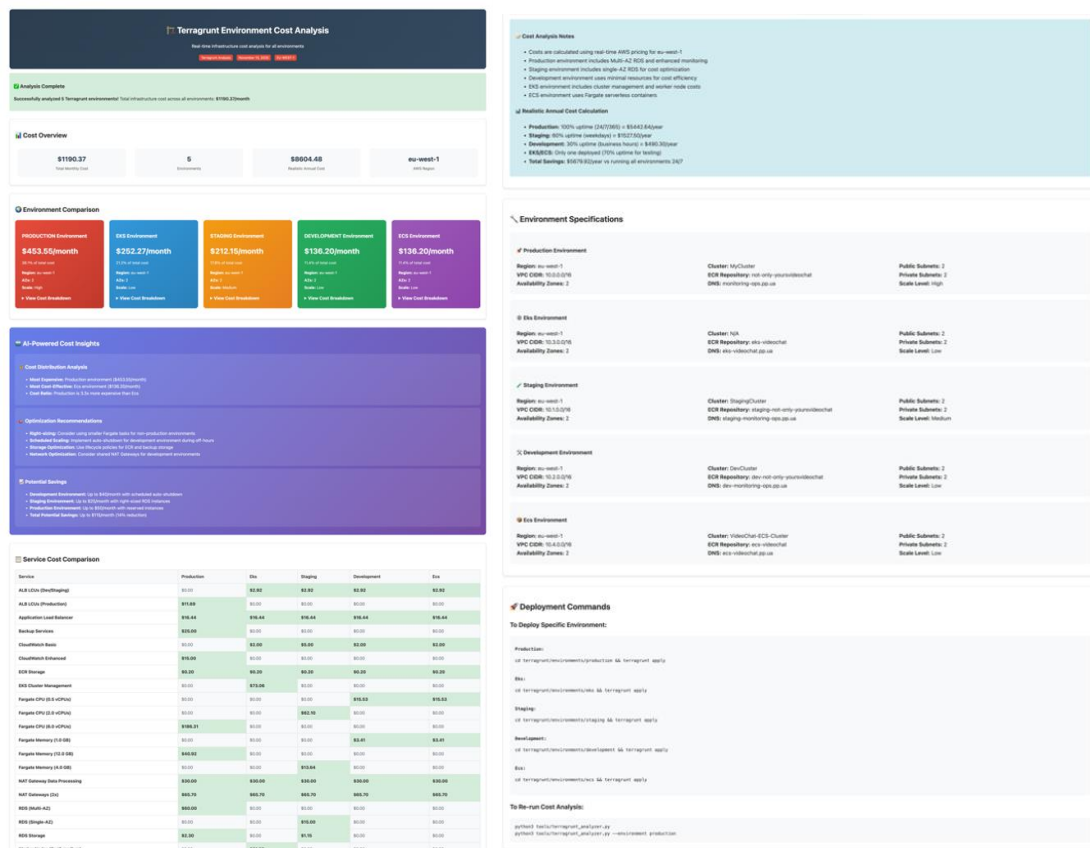


Figure 5. Web-based interface of the analysis results

Practice has shown another advantage of the proposed approach. Within the framework of one process, it was possible to combine current tariff data, Infrastructure as Code analysis, scenario modeling and cost estimation taking into account automatic scaling. It helped the results be more consistent to what the real infrastructure behaves. Unlike static calculator or an expert. And at the same time it's obvious where the model has its limitations. The verification was carried out mainly on a stateless application that worked in one region. Under such conditions, the behavior of horizontal scaling can be described relatively simply, using the load intensity as the main parameter.

In more complex cloud environments, this is no longer enough. The performance begins to be dominated by data replication, storage coherence and latency between different regions, besides increased transmission costs for the traffic. This effect becomes more pronounced in case of stateful services and geo-distributed clusters.

We do not consider these features as a disadvantage of the proposed methodology. Rather, they outline the limits of its current validation and show in which direction the model should be developed further for more complex multi-regional cloud infrastructures.

4. CONCLUSION

We have presented a methodology for cloud architecture comparison under different workloads. Our model integrates Infrastructure as Code, dynamic pricing of the cloud, workload scenarios, autoscaling, costing and performance estimation into a single analytic workflow that models realistic cloud deployments rather than simplified models of infrastructure. Real deployments could be analyzed with real infrastructure setup rather than abstract models. In the experiment, the historical data of production cloud application are used. The accuracy of forecast achieved 94.6% in average and its standard deviation was 1.2% with the maximum result 97.1% for EKS Medium. Local Docker deployment has achieved the overall accuracy of 98.6% with cost that was little changing over time, and in ECS Small a deviation of 5.2% was noted for the short duration of a traffic burst. The application of this method reduced manual work on comparing architectures.

The configurations of Terraform and Terragrunt are analyzed by means of parsers, avoiding explicit entry of parameters. Data about prices were acquired automatically by the APIs from providers, and the 24 hours cache reduced pricing request from 3.2 sec to 0.03 sec on average. Within 2 to 3 seconds 3 architectures and 4 scenarios were already compared. Stateless application that is deployed in a single region is analyzed in the presented experiment. The extended models for the network latency between different regions or replication, consistency and synchronization between regions need to be studied for such applications.

Future work will involve adapting the model for more complex, distributed, and multi-cloud applications which include requirements for replication and consistency and inter-cloud traffic.

REFERENCES

- [1] J. Yang and A. Abraham, 'Analyzing the Features, Usability, and Performance of Deploying a Containerized Mobile Web Application on Serverless Cloud Platforms', *Future Internet*, vol. 16, no. 12, p. 475, Dec. 2024, doi: 10.3390/fi16120475.
- [2] A. Alkhatib, A. Shaheen, and R. N. Albustanji, 'A Comparative Analysis of Cloud Computing Services: AWS, Azure, and GCP', *Int. J. Comput. Digit. Syst.*, vol. 18(1), pp. 1–15, Aug. 2024, doi: 10.12785/ijeds/1571111846.
- [3] L. H. Bayzid, T. S. Kar, M. T. Islam, M. S. Islam, and F. Ahmed, 'Defending the Distributed Skies: A Comprehensive Literature Review of the Arena of Multi-Cloud Environment', *Future Internet*, vol. 17, no. 12, p. 548, Nov. 2025, doi: 10.3390/fi17120548.
- [4] 'StormForge Case Study', Amazon Web Services, Inc. Accessed: Apr. 26, 2026. [Online]. Available: <https://aws.amazon.com/solutions/case-studies/stormforge-case-study/>
- [5] J. Sica and A. Lawson, 'CNCF Annual Cloud Native Survey. The infrastructure of AI's future', The Linux Foundation, Jan. 2026. Accessed: Apr. 26, 2026. [Online]. Available: https://www.cncf.io/wp-content/uploads/2026/01/CNCF_Annual_Survey_Report_final.pdf
- [6] E. Özdoğan, O. Ceran, and M. T. Üstündağ, 'Systematic Analysis of Infrastructure as Code Technologies', *Gazi Univ. J. Sci. Part Eng. Innov.*, vol. 10, no. 4, pp. 452–471, Dec. 2023, doi: 10.54287/gujisa.1373305.
- [7] A. Elhabbash, F. Samreen, J. Hadley, and Y. Elkhatib, 'Cloud Brokerage: A Systematic Survey', *ACM Comput Surv*, vol. 51, no. 6, p. 119:1-119:28, Jan. 2019, doi: 10.1145/3274657.
- [8] N. Thakur, A. Singh, and A. L. Sangal, 'Cloud services selection: A systematic review and future research directions', *Comput. Sci. Rev.*, vol. 46, p. 100514, Nov. 2022, doi: 10.1016/j.cosrev.2022.100514.
- [9] J. Araujo, P. Maciel, E. Andrade, G. Callou, V. Alves, and P. Cunha, 'Decision making in cloud environments: an approach based on multiple-criteria decision analysis and stochastic models', *J. Cloud Comput.*, vol. 7, no. 1, p. 7, Mar. 2018, doi: 10.1186/s13677-018-0106-7.
- [10] G. Ilieva, T. Yankova, V. Hadjieva, R. Doneva, and G. Totkov, 'Cloud Service Selection as a Fuzzy Multi-criteria Problem', *TEM J.*, pp. 484–495, May 2020, doi: 10.18421/TEM92-09.

- [11] G. Fragiadakis, A. Tsadimas, E. Filiopoulou, G. Kousiouris, C. Michalakelis, and M. Nikolaidou, 'Cloud PricingOps: A Decision Support Framework to Explore Pricing Policies of Cloud Services', *Appl. Sci.*, vol. 14, no. 24, p. 11946, Jan. 2024, doi: 10.3390/app142411946.
- [12] M. Menzel, M. Schönherr, J. Nimis, and S. Tai, '(MC2)2: A Generic Decision-Making Framework and its Application to Cloud Computing', Dec. 16, 2011, *arXiv*: arXiv:1112.1851. doi: 10.48550/arXiv.1112.1851.
- [13] V. Prakash, A. Kumar, M. Shahid, L. Garg, and S. Bawa, 'An efficient model to estimate and optimise the cloud migration costs from on-premises web apps', *Discov. Comput.*, vol. 28, no. 1, p. 151, Jul. 2025, doi: 10.1007/s10791-025-09666-3.
- [14] V. Kozlovskiy, N. Yakymchuk, Y. Selepyna, S. Moroz, and A. Tkachuk, 'Development of a Modified Method of Network Traffic Forming', *Inform. Autom. Pomiary W Gospod. Ochr. Śr.*, vol. 13, no. 1, pp. 50–53, Mar. 2023, doi: 10.35784/iapgos.3452.
- [15] E. Jafarnejad Ghomi, A. M. Rahmani, and N. N. Qader, 'Applying queue theory for modeling of cloud computing: A systematic review', *Concurr. Comput. Pract. Exp.*, vol. 31, no. 17, p. e5186, 2019, doi: 10.1002/cpe.5186.
- [16] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, 'Adaptive, Model-driven Autoscaling for Cloud Applications', presented at the 11th International Conference on Autonomic Computing (ICAC '14), Philadelphia, PA, Jun. 2014, pp. 57–64. [Online]. Available: <https://www.usenix.org/system/files/conference/icac14/icac14-paper-gandhi.pdf>
- [17] A. Q. Khan, M. Matskin, R. Prodan, C. Bussler, D. Roman, and A. Soyly, 'Cloud storage cost: a taxonomy and survey', *World Wide Web*, vol. 27, no. 4, p. 36, May 2024, doi: 10.1007/s11280-024-01273-4.

BIOGRAPHY OF AUTHORS



Valeriy Kozlovskiy   graduated from the Kyiv Higher Military Aviation Engineering School (Ukraine) in 1992 with the qualification of Radio Engineer. In 1994, he received a PhD in Armament and Military Equipment from the Kyiv Air Force Institute. In 2013, he earned a Doctor of Science degree from the State University of Information and Communication Technologies (Kyiv, Ukraine). In 2014, he was awarded the academic title of Professor.

Since 2014, he has served as Head of the Department of Information Protection at the Educational and Scientific Institute of Information and Diagnostic Systems of the National Aviation University. Since 2018, he has held the position of First Vice-Rector of the National Aviation University. Since 2020, he has been Head of the Department of Information Protection at the National University "Kyiv Aviation Institute" (Kyiv, Ukraine).

His research interests include intelligent decision-support systems, information protection management, and end-to-end control and optimization in information and computing networks.

You can contact him by e-mail: vvkzeos@gmail.com



Yuriy Lysetskyi   graduated from the S.M. Kirov Kyiv Higher Anti-Aircraft Missile Engineering School in 1983 with a degree in radio engineering systems. He served in the Trans-Baikal Military District, the Syrian Arab Republic, the Research Institute of Air Defense of the Land Forces, and the Central Research Institute of the Armed Forces of Ukraine. In 2001, he graduated from Kyiv National Economic University named after Vadym Hetman, specializing in International Economics. In 2016, he completed doctoral studies at the Institute of Problems of Mathematical Machines and Systems of the National Academy of Sciences of Ukraine.

He is Doctor of Technical Sciences, Professor of the Department of Information Protection Systems, National University "Kyiv Aviation Institute," Kyiv, Ukraine. He is the author of more than 100 scientific publications, including four monographs.

His research interests include system analysis, complex systems research, decision-making and management processes, information technologies, and AI-based system protection.

You can contact him by e-mail: Yurii.Lysetskyi@snt.ua



Nikita Bakun  holds a Master's degree in Software Engineering from the Igor Sikorsky Kyiv Polytechnic Institute and a Master's degree in Cybersecurity from the National Aviation University, Kyiv, Ukraine. He is currently a PhD student at the National University "Kyiv Aviation Institute," pursuing research in applied machine learning infrastructure and security.

He works as an MLOps Engineer at AJAX Systems, where he focuses on designing and maintaining scalable machine learning infrastructure and deployment pipelines.

His research interests include MLOps, machine learning lifecycle management, deployment and monitoring of ML models, Infrastructure as Code (IaC), the use of Terraform in MLOps pipelines, cloud-based ML systems, and machine learning infrastructure security.

You can contact him by e-mail: darty.vis@gmail.com



Diana Kozlovskaya   is a student at the National University “Kyiv Aviation Institute,” Kyiv, Ukraine, majoring in Cybersecurity. Her academic training focuses on information security, data protection technologies, and modern cyber defense methods.

Her research interests include neurobiology, cybernetics, machine learning, and data analysis, with a particular focus on interdisciplinary approaches to intelligent systems and human-machine interaction.

You can contact her by e-mail: 8542876@stud.kai.edu.ua



Natalia Yakymchuk   received a Master's degree in Automated Control of Technological Processes from Lutsk State Technical University (Ukraine) in 2005. In 2023, she received a PhD from the National Aviation University (Kyiv, Ukraine) in the field of telecommunication systems and networks.

She is currently an Associate Professor at the Department of Electronics and Telecommunications of Lutsk National Technical University (Lutsk, Ukraine). Her research focuses on diagnostics and control of telecommunications network states and congestion management.

You can contact her by e-mail: n.yakymchuk@lntu.edu.ua