Servitization of Customized 3D Assets and Performance Comparison of Services and Microservices Implementations

Angel Ruiz-Zafra, Janet Pigueiras-del-Real, Manuel Noguera, Lawrence Chung, David Griol Barres, Kawtar Benghazi

Abstract—3D models (or assets) that are present in many of modern software applications are first modeled by graphic designers using dedicated computer graphic tools and then integrated into such software applications or apps by software developers. This simple workflow/procedure requires developers to have a basic grounding in computer graphics, since 3D engines, libraries and third-party software are needed for this kind of integrations. Oftentimes, 3D designers are also required to customize or produce versions of a 3D model and thus, they must re-model all the assets before they are returned back to the developers for integration into the applications. This procedure also occurs whenever a modification or customization is requested. One possible significant improvement to this traditional, poorly automated workflow is to use services-oriented technology and features servitization to carry out the customization of 3D assets on-demand. In this paper, we introduce µS3D, an open-source microservices-based platform designed to support features relating to the customization of 3D models. µS3D not only enables 3D assets to be customized without the need for computer graphic tools or designers, but also allows 3D models to be visualized through web technologies (e.g. HTML, Javascript and web component to visualize and interact with 3D models), thereby avoiding the development of computer graphics libraries or components in final software products. The paper describes the elements that µS3D comprises, explains how it works and presents a series of load tests to compare the performance (time consumption, CPU and memory utilization) of µS3D when implemented and deployed as a microservices platform against a monolithic-based implementation, showing similar results with a low number of users (and requests) but reducing, on average, 64.32% the response time in the microservice-based implementation for a large number of users; reducing CPU utilization on microservice-based implementation and remaining the memory usage more or less constant in both implementations.

Index Terms—3D models, graphics, services, microservices, customization, µs3d, architectures.

1 Introduction

3D models play a pivotal role in various industries and fields due to their ability to provide highly realistic visual representations, enhance user experiences [1], [2], improve communication and understanding, enable simulations and training, support virtual and augmented reality experiences, drive marketing and sales efforts, etc. Their versatility and impact make 3D models indispensable tools in today's digital age, enabling innovation, improved communication, and breakthrough discoveries across numerous domains [3], [4].

Customization capabilities are relevant in these applications as users frequently seek to personalize the objects they interact with and this ability to customize enhances user engagement and satisfaction [5]. However, it is impractical to expect designers to create or anticipate every possible combination of customization options and preload them in applications. As the number of customization options increases, the task becomes exponentially challenging. For instance, let us consider a 3D object that can be painted in four different colors and animated in five different ways. This would require producing 20 unique 3D models (4 colors x 5 animations). As a result, this approach becomes unfeasible and impractical.

The integration of 3D models and the development of 3D customization functionalities in software applications require a knowledge base that goes beyond traditional programming skills. It demands a deep understanding of computer graphics, and the underlying principles of three-dimensional representations [6]. For instance, a developer aiming to incorporate a 3D model viewer in a web application would need to utilize WebGL or Three.js, along with their APIs and libraries, to handle interaction. Additionally, they would need to employ techniques such as texture mapping, lighting, and shading to achieve realistic visual effects. Furthermore, knowledge of 3D file formats like OBJ or FBX along with parsing algorithms, would be necessary for loading and manipulating the 3D models [7].

Furthermore, it is important to note that the specific technologies and concepts required may vary depending on the target platform (e.g., web, desktop, mobile) and the chosen 3D graphics library or framework being used in the software development process [8], [9].

Nonetheless, developers can overcome the steep learn-

- *Angel Ruiz-Zafra, Manuel Noguera, David Griol Barres and Kawtar Benghazi are with the Departament of Software Engineering - ETSIIT - University of Granada (Spain).*
  E-mail: {angelr,mnoguera,dgriol, benghazi}@ugr.es
- *Janet Pigueiras is with the Departament of Physics and Condensed Material - University of Cadiz (Spain)*
  E-mail: janet.pigueiras@uca.es
- *Lawrence Chung is with the Departament of Computer Science - The University of Texas at Dallas (UT Dallas) (United States)*
  E-mail: chung@utdallas.edu

ing curve associated with mastering various graphics technologies and frameworks by providing them with standardized and simplified approaches to 3D model integration. Servitization of functionalities and their encapsulation behind accessible API's, particularly web services, has proven to be a successful approach in many software application domains in the past, but still underexplored when it comes to 3D manipulation [10].

The design and encapsulation of functionalities, following the SOA (Service-Oriented Architecture) approach, would not only simplify the manipulation of 3D models through platform-independent technologies and standards. In modern deployments, it could also leverage the power of microservices and containerization, specifically using tools like Docker and Kubernetes, to unlock numerous benefits.

By adopting a microservices architecture, the integration of 3D models into software applications becomes highly modular and scalable. Each microservice can be dedicated to a specific aspect of 3D model processing, such as animation or physics simulation, enabling easier development, testing, and maintenance of individual components [11]. This decoupling of functionalities allows for independent scaling, deployment, and updates, resulting in enhanced flexibility and faster time-to-market.

Containerization, facilitated by Docker, plays a crucial role in efficiently deploying and managing microservices. Containers provide a lightweight and consistent runtime environment that encapsulates the required dependencies, ensuring a seamless deployment process across various platforms. Developers can package each microservice, along with its dependencies, into self-contained units, ensuring portability, scalability, and easy replication across different environments [12].

To further enhance the management of microservices, Kubernetes, an orchestration platform for containerized applications, automates the deployment, scaling, and load balancing of containers. It ensures high availability and efficient resource utilization. Kubernetes also offers features like self-healing, rolling updates, and service discovery, enabling the seamless integration of microservices into a cohesive system.

This paper presents *µS3D*, a microservices-based platform devised to support tasks relating to the manipulation of 3D models (e.g., customization of parts of 3D models, texture mapping and animations) on the fly without requiring to develop complex 3D manipulation libraries or embedding heavy 3D engines in software applications as well as to know how to manipulate 3D objects using such engines.

µS3D comprises three core elements: *a microservices platform*, which is a set of microservices [13] supported by Minikube (single-cluster version of Kubernetes) and Docker; *a wrapper*, which provides a high-level interface for interaction with an independent computer graphic engine; and *the 3D model fingerprint*, which is a cache-based system to enhance performance in terms of storage and response time to generate customized 3D models on demand. It has also been implemented a monolithic version of *µS3D* in order to compare both implementations (i.e., microservice-based and monolithic), in terms of performance by conducting a number of load and stress tests.

The main contributions of this work are:

- To enable the customization of 3D models, eliminating the need for a 3D designer once the models are generated.
- it is a ready-to-use platform that allows developers of any background, even without technological expertise on 3D assets or engines, to easily invoke functionalities for customizing 3D models.
- while the integration and customization of models require specific technology depending on the platform, our proposal (named *µS3D*), is platform-agnostic.
- there are two implementations available, monolithic and microservices-based, that can be used depending on each project features.
- a comparison between both implementations to help determine the most suitable one in terms of performance, CPU and memory average use is provided.
- *µS3D* is currently available as an open-source platform so that all the source code relating to both implementations, the datasets used in the load-testing and performance stage and a video demonstration of µS3D working can be found in the repository described in the *Supplementary Materials* section.

The remainder of the paper is structured as follows: Section 2 presents related work and details other approaches that improve the classical integration approach; Section 3 describes µS3D and explains its three core elements (i.e. microservices, wrapper and cache-based system - 3D fingerprint-); Section 4 describe the workflow of our proposal and an illustrative example. For the evaluation of our proposal, Section 5 describes the Methods and Section 6 illustrates the results and draw the conclusions. Finally, Section 7 discusses the proposal and Section 8 outlines our conclusions and future work.

2 Related Work

In conventional practices, designers rely on specialized software to create 3D assets, which are subsequently integrated into applications by developers [15], [19], [26]. Improvements have been made in recent years to enhance this approach by focusing on the release of custom software to support the modeling or creation of 3D assets. Such software solutions usually focus on a specific application domain, such as building construction and creating characters, among others [15]. Additional software has been developed as add-ons or plugins for 3D computer graphics tools, such as *MB-LAB*[1]and *AVATAR*, a Blender add-on that enables fast creation of 3D human models [14].

The purpose of this specialized software is to empower graphic designers with the ability to create 3D models using a lower level of expertise compared to other 3D tools like Blender, 3D Studio Max and Maya. However, the integration process for developers remains the same, where the graphic designer sends the 3D model (which may comprise one or several files) to the developer who then determines how to integrate it into the application software. Additionally,

1. https://mb-lab-community.github.io/MB-Lab.github.io/

TABLE 1: Comparison of the most relevant works for 3D customization

Category	Project	Any 3D model	3D Modeling Knowledge	3D Technology Knowledge (Integration)	Fully customization	Additional Knowledge Required
3D Graphic Tool	**AVATAR** [14]	No	Yes	Yes	Yes	Blender Plugin
	MakeHuman [15]	No	Yes	Yes	Yes	No
	MB-LAB (Footnote 1)	No	Yes	Yes	Yes	Blender addon
Semantic-based approaches	**MICC** [16]	Yes	Yes	Yes	Yes	Ontologies, Semantic Web
	3D with semantic queries [17]	Yes	Yes	Yes	No	Ontologies, 3D modeling tool
Novel software solutions	**SGToolKit** [18]	No	No	Yes	No	Custom software
	Web3D Customization [19], [20], [21]	No	No	No	No	Custom software, Web programming
Use of Artificial Intelligence	**AvatarClip** [22]	No	No	Yes	No	Skinned Multi-Person Linear Model
	Rodin [23]	Yes	No	Yes	No	Diffusion models
	AlteredAvatar [24]	No	No	Yes	No	Contrastive Language-Image Pre-Training (CLIP) neural network
	Style3DAvatar [25]	No	No	Yes	No	Generative Adversarial Networks (GAN)

the graphic designer is responsible for customizing the 3D model according to end-user needs by means of a separate, offline procedure for making any necessary modifications and for sending the 3D asset back to the developer. This usually requires the software to be re-built and forces end-users to update to a new version each time a change in the 3D asset is introduced.

Other improvements to enhance this classical approach have come from computer science, where several projects and solutions based on IT technologies have been proposed in order to simplify the integration, generation and management of 3D models. These projects and research can be classified into three different categories: *semantic-based approaches* to represent or customize 3D content, *novel software-based solutions* to simplify the customization of 3D content, and *AI-based solutions* to generate customized 3D models from text or images.

In the first category, semantic-based approaches generally use ontologies. For instance, the project presented by the authors in their article [16] introduces *MICC (Method of Inference-based 3D Content Creation)*, a method to create 3D content using OWL ontologies while hiding technical details which are specific to 3D graphics through a mapping between domain-specific concepts and graphic specific contents.

In the same category, other projects go one step further by not merely representing 3D contents through non-traditional 3D formats, but also by introducing the possibility of customization. A prominent example is proposed by Flotynski et al. [17], who propose that 3D contents be customized through semantic queries represented in XML (*Extensible Markup Language*).

In the second category of novel software applications, a number of projects have recently appeared to simplify the use and integration of 3D models through handy *ad-hoc* applications. For instance, *SGToolKit* [18] is a toolkit for customizing gesturing for human-like 3D models, i.e. avatars. The application was developed so as to enable the customization of the 3D model animations (rather than parts of the model or textures) to simulate humanoid gestures from the voice commands of a human interlocutor. *SGToolKit* uses text-to-speech techniques together with deep neural networks in order to classify the body posture to be visualized in the 3D character.

In this same category, a current trend is the use of web technologies to handle or customize 3D content in order to reduce as much as possible the knowledge for manipulating 3D assets. For instance, Piao et al. [19] and Liu et al. [20] present solutions supported by HTML and JavaScript as technologies for visualizing and customizing 3D content. A web-based solution to manage 3D content is presented by Nicolaescu et al., [21] who describes an approach to support real-time collaboration to annotate 3D objects on the web through a microservice architecture. In this case, the customization primarily focuses on the metadata of the 3D object for medical purposes.

Other two illustrative examples in this category are described in [27] and [28], presenting web services-based tools to generate customized 3D models related to genomics and protein structures.

Lastly, and being the third category and the most innovative one, other projects primarily driven by major companies such as Meta® and Microsoft® make use of AI-based technologies to generate customized 3D models. Most prominent proposal is *Rodin*, a generative model for custom 3D avatars through neural networks [23]. *Rodin* enables the customization of 3D avatars from images or text using Blender pipeline along with an AI-based model, trained with 100K 3D avatars. Some other renowned projects oriented to 3D generative models are *AlteredAvatar, AvatarClip* and *StyleAvatar3D* [22], [24], [25].

The three presented categories (semantic-based, novel software applications and AI-based solutions) significantly improve the customization and integration of 3D models, as well as enhance and simplify the tasks performed by graphic designers and developers. However, these approaches present several drawbacks:

- Solutions aimed at simplifying the creation of 3D models by graphic designers reduce total project times and also the response time to provide the required customized 3D models, but this does not significantly affect developers endeavours, since they still have to learn 3D-management technologies.
- Semantic-oriented approaches use ontologies, and this also requires an in-depth knowledge of conceptual modeling languages and related APIs to create, query and delete ontological representations.
- The other solutions based on dedicated applications for creating or customizing 3D models are very limited, because they are usually ad-hoc applications which are created for a specific purpose.
- Manual intervention by developers is required when using these *ad-hoc* applications, which is part of the integration time and, therefore, increases the time required for the development process.
- Web-based approaches for customizing 3D models also require considerable knowledge about graphic design, because developers must learn about 3D file formats and how to handle the parts or components of each 3D asset.
- The integration of 3D models by means of web-based technologies still requires a great deal of coding and learning about dedicated 3D libraries.
- The AI-based approaches are still in their infancy, so these solutions are far from being used by the general public or developers, requiring knowledge of different machine learning techniques such as diffusion models or Generative Adversarial Networks (GAN).
- The AI-generated 3D assets are created according to the trained model, so fully customization of assets (i.e. personal textures, animations or assemblies) is not supported.

Table 1 summarizes the most relevant works in the field and the characteristics of each one in terms of 3D model generation.

In this paper, we present *μS3D*, a microservices-based platform to ease the access and invocation of customization functionalities for 3D models. Unlike other solutions, *μS3D* enables developers to address the integration of 3D models into software applications without any prior knowledge of 3D computer graphics and tools. The remainder of the paper describes *μS3D* in detail.

3 *μS3D*: SERVICES AND MICROSERVICES FOR MANAGING 3D ASSETS

μS3D has been designed to streamline the integration process of 3D objects for developers, eliminating the necessity of relying on specialized 3D computer tools. Moreover, it eliminates the tedious task of making post-design changes for 3D designers once the models have been fully designed. μS3D comprises three core elements (see Figure 1):

1) a *microservices platform*,
2) a *wrapper* to interact with a graphic engine,
3) and the *3D model fingerprint*, which is a cache-based system to improve performance in terms of storage and response time.

These three core elements are described in the remainder of this section.

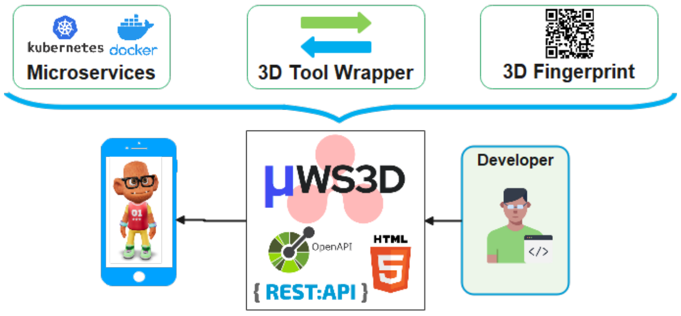


Fig. 1: μS3D-based core elements

3.1 Microservices Platform

μS3D offers eight microservices, each dedicated to providing one of the following functionalities related to 3D asset customization:

- to list the set of available 3D models in a series of 3D files
- to list the set of model animations
- to list the set of a 3D model parts
- to hide (or make not visible) a series of 3D model parts
- to show (i.e., make it visible) a series of 3D model parts
- to map a color to a series of parts
- to map an image texture to a series of parts
- and, finally, to obtain the customized generated 3D model.

The orchestration of these services enables developers to include functionalities for the customization of the three main parts (assemblies/skeleton, textures, animations) of 3D models by servitizing a computer graphic engine. The API of these microservices is described using the OpenAPI 3.0 specification[2], and a guideline file can be found in the repository referenced in the *Supplementary Materials* section.

The use of these services for customizing 3D models implies that at some point, a computer graphic software tool is necessary in order to process requests from these microservices (e.g., to hide certain assemblies or list the animations of a model) and modify a stored 3D model (in a file or set of files). This interaction between the microservices and the graphic engine is conducted through the second component of the *μS3D* platform, that is, the *wrapper*.

3.2 Wrapper

A custom *wrapper* has been developed to bridge the gap between the microservices and the 3D computer graphic

2. https://www.openapis.org/

engine. This *wrapper* implements the low-level code required to interact with the 3D engine interface to provide the functionalities relating to customizations of the 3D model. Although many different computer graphic tools are available for 3D modeling, we opted for Blender®[3] not only because it is free (3DS Max® and AutoDesk Maya® require paid licenses), open source and multiplatform (which makes it accessible to a wide range of users), but mainly because Blender provides a high level API, based on Python, to access its processing engine.

When a microservice is invoked by the end-user software application, a new instance of Blender is launched. Several parameters are used to specify the operation to be conducted (e.g., hide/show certain parts, change a texture, etc.), the name of the model, and the payload of the request (e.g., the color of the new texture, the name of the assemblies). The wrapper is executed by the Blender engine and performs the operation(s) on the model using the payload, thereby generating a new customized model.

Although the wrapper returns the path of the generated model to the microservice, the microservice returns an HTML fragment to the end-user in order to display the custom 3D model online. This allows any application to visualize the 3D model efficiently without the need for additional software because most current platforms and technologies support HTML code embedding.

The *wrapper* also supports the main 3D file formats compatible with Blender, i.e., Blender native files, FBX (FilmBoX), GLB/GLTF (Graphics Language Transmission Format) and COLLADA. This significantly improves the integration of µS3D and enables 3D models to be customized in the most common 3D file formats. Furthermore, in order to ensure the integration of any type of engine, all customized 3D models are exported and released in GLB/GLTF format, which is basically a description of the 3D asset using standard notation and which is supported by web technology (HTML5, JavaScript and third-party libraries).

3.3 3D Model Fingerprint

No matter how large they are, 3D assets have a limited number of animations, assemblies (or complements), and set of textures. Every possible permutation or combination of these elements would generate a high number of customized 3D models (for example, a model with 25 assemblies/complements, one possible texture for each one and 10 animations could result in up to 25x25x10=6250 possible customized models). However, it is not uncommon, especially with simple 3D models, for several users to select the same customizations for the same 3D model, which may result in duplicating unnecessarily thestorage space for 3D models.

µS3D therefore includes a cache-based system of *3D model fingerprints*, which represents a self-contained description of the 3D model together with any customization that has been requested. Rather than containing the 3D asset, the 3D model fingerprint contains a light-weight description file that includes the name of the 3D asset, the selected parts and texture values of each customized complement, the selected animation, and the path to the final generated 3D custom asset. In this way, if the user requests a specific customized model, each microservice checks whether the requested features for a model match any 3D model fingerprint already generated and stored. If there is a 3D model fingerprint match, the HTML code with the embedded 3D asset is returned to the end-user, thereby avoiding running a new Blender instance and the different tasks carried out by the *wrapper* already described. This not only reduces the response time perceived by the user, but it also avoids the generation of all possible 3D assets at once in certain scenarios or applications.

3.4 *µS3D*: Design and Development

The *µS3D* platform has been designed to provide features by means of microservices which can be accessed by developers, designers and anyone who may be interested in incorporating them into their software. The *µS3D* microservices implementation architecture is displayed in Figure 2.

This implementation has been developed using two core technologies: Kubernetes[4] and Docker[5]. The different services have been organized around *business capabilities* relating to *assemblies, textures* and *animations*. Each business capability includes an independently deployed system running on Docker, which includes the different web services that represent the various features. Each business container is deployed into a *"POD"* (i.e., the smallest deployable unit of computing that you can create and manage in Kubernetes, and which encapsulates and manages multiple containers) and this is replicated n-times to provide scalability. In this implementation, each business container has been replicated three times. Each one is a separate computing unit and can accept incoming requests independently. A set of replicas of a specific POD is called a service.

The eight microservices introduced in subsection 3.1 have been grouped and deployed through five Kubernetes services, namely: *assemblies, animations, textures, utils* and *Blender*. Each Kubernetes service comprises three PODs, each running a Docker image with an independent server to access the microservices located in the POD, and which also serve functionalites other than those related to 3D models manipulation. Likewise, each Kubernetes service has an endpoint (e.g., */assemblies, /animations, /textures, /utils, /blender*) and a load balancer that determines where to forward requests among the different PODs available. The load balancer used is *MetallB*[6]. Furthermore, at the top of the architecture (above all the load balancers), there is a light server deployed that acts as a shared access point with two main goals: the first is to distribute the requests using an additional load balancer, and the second is to forward the requests to the proper entry point. Each service (i.e., a set of a POD's replicas) represents an independent deployment supported by Docker and accessed through a Flask server. Although each server provides an internal IP address, in Kubernetes access between services is through the service name (µWS3D-Assemblies, µWS3D-Animations, etc). In this way, when an end-user application requests something relating to assemblies (*/assemblies*), the

3. https://www.blender.org/

4. https://kubernetes.io/
5. https://www.docker.com/
6. https://metallb.universe.tf

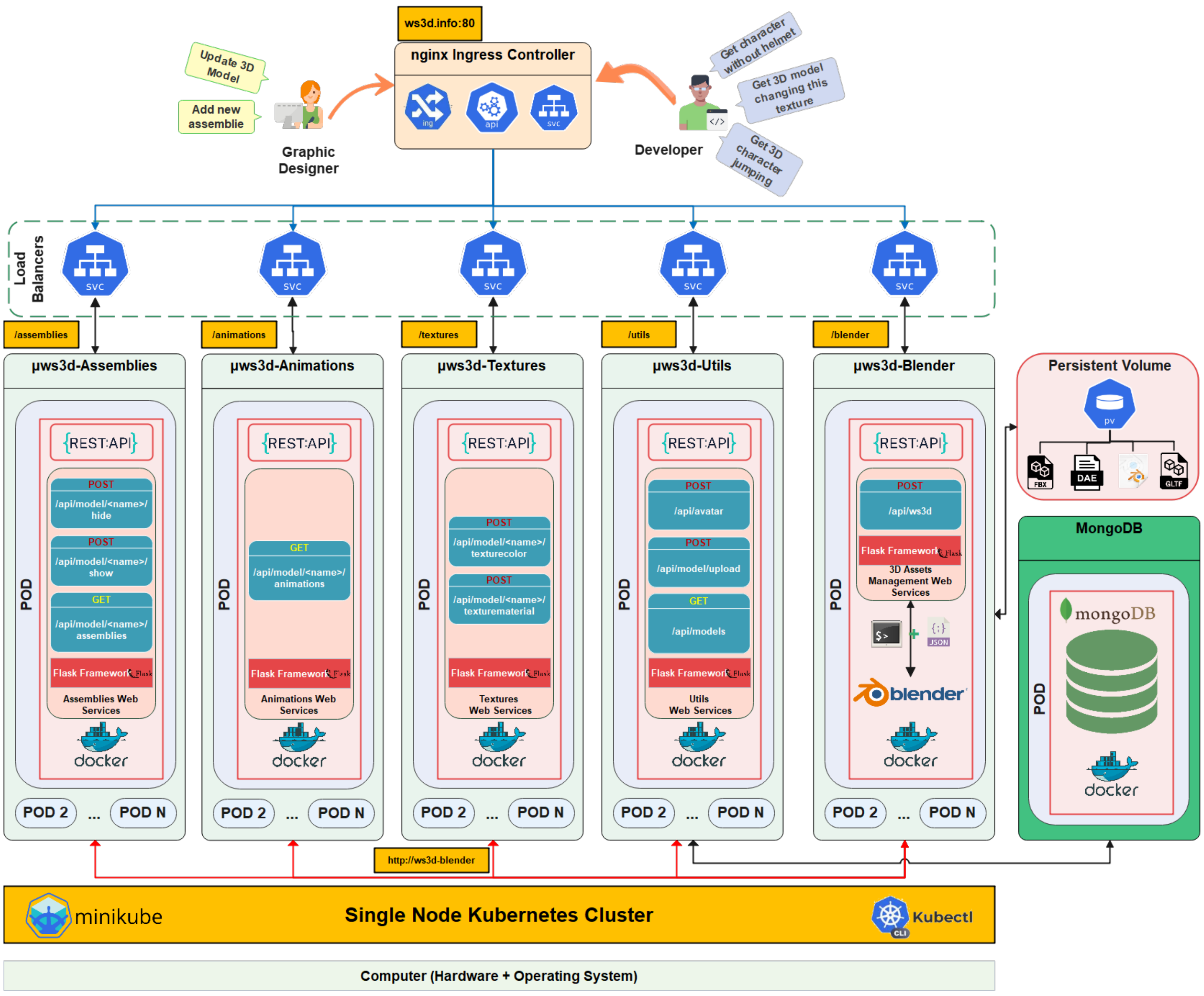


Fig. 2: *µS3D* platform architecture

microservices implemented in the µWS3D-Assemblies service are invoked. This forwarding between the request (e.g., *http://ws3d.info/assemblies*) and the service where the PODs with the microservices are running (*µWS3D-Assemblies*) is performed through the so-called *ingress*, an API object that manages external accesses to the services in a cluster.

Besides Kubernetes services, there is also a sixth container with a database management system (in our case, it is MongoDB) to manage 3D models fingerprints and also a persistent volume mapped to a directory inside the running Minikube instance to store the 3D model files.

This microservices architecture greatly enhances scalability since a large number of requests (or consumption services) are not managed by the same processing unit and endpoint (as in a monolithic implementation). Load balancers distribute requests to the different PODs which are running in independent processing units, and the *ingress* forwards each external access to the corresponding microservice. Although a processing unit could also be a dedicated computer, this current implementation has been developed using Minikube[7], a local Kubernetes deployment based on a cluster with a single node.

4 *µS3D* Workflow and Application Example

4.1 Workflow

Figure 3 illustrates the workflow of *µS3D*, showcasing how our proposal functions using an example model. Additionally, we have included a video demonstration in the *Supplementary Material* for further clarity and visualization (see Section 8).

Thanks to *µS3D* microservices, graphic designers can upload new models or update existing ones (Step 1) that developers or other types of end-user applications can use. End-user applications are able to manage the different features provided relating to the animations, textures and assemblies/complements (1-6) by using microservices. Models are exported in GLTF (a standard file format for three-dimensional scenes and models) in order to enhance

7. https://minikube.sigs.k8s.io/

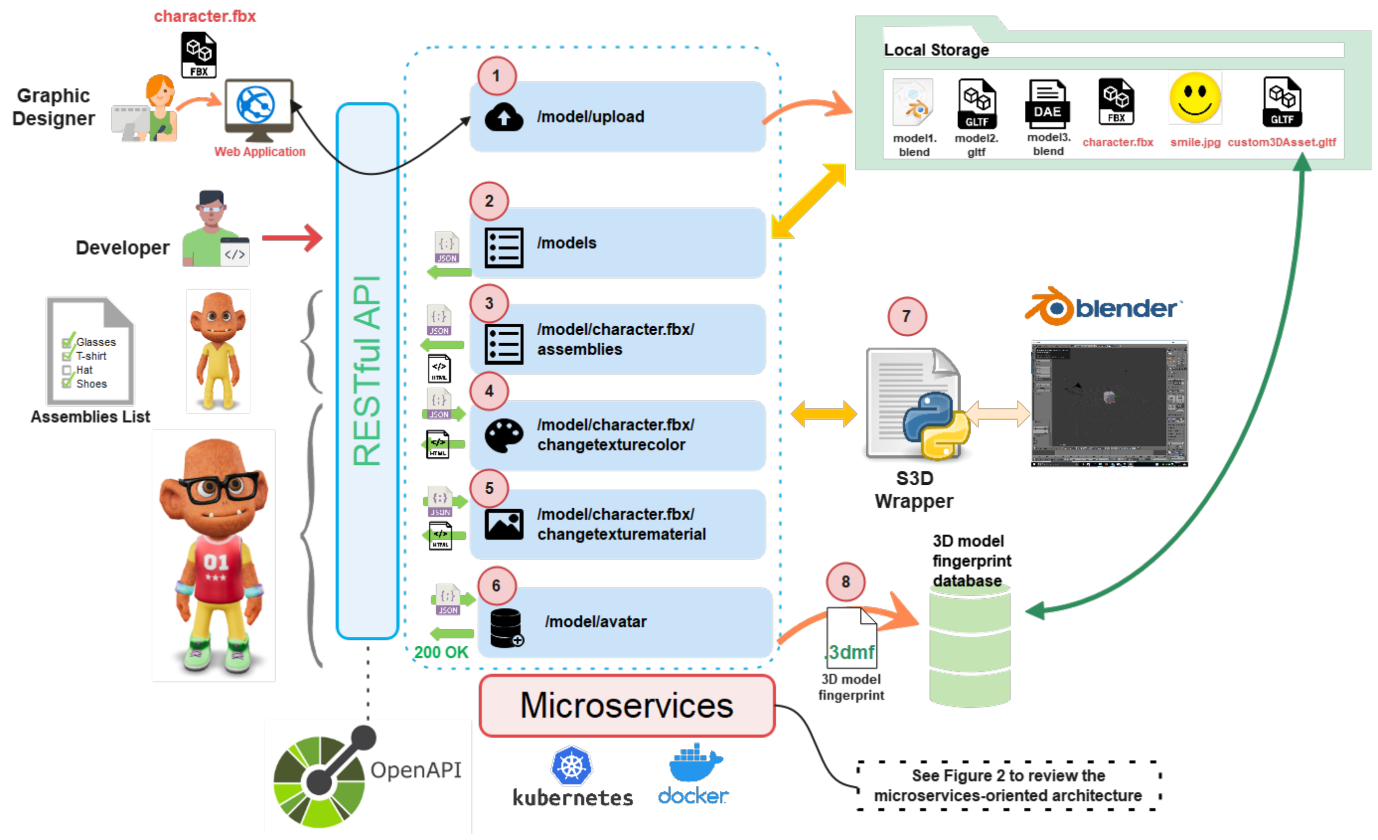


Fig. 3: 3D Customization example in *μS3D*

their visualization and integration. This means that, if the format of the original model container is different from GLTF (for instance, FBX or Blend), the wrapper will export the model with the modifications as a GLTF file. This is performed in this way to protect the original uploaded model, thereby preventing the changes in a specific 3D model from affecting the customization of 3D models by other users. For example, the activity of listing the parts of a 3D model (represented by step number 3 and the file *character.fbx* in Figure 3) entails launching a new Blender instance, importing the fbx file model, looping through the collections to find the different parts and transforming the original model (*fbx*) into a new GLTF model (i.e. character_custom_1.gltf), thereby providing the user with a list of available parts, along with the corresponding GLTF model generated embedded in an HTML code snippet. This *modus operandi* is reproduced through the different microservices until the end-user application receives the order to generate the final, customized 3D model (8) (*customized3DAsset.gltf*) and any previous intermediate 3D customized model is removed.

4.2 Synthetic *μS3D*-based solution for the customization of 3D models

Besides the workflow described above, which also illustrates the customization of a 3D model represented in 2D figure, we have prepared a synthetic web platform, consisting of a toy web application and a back-end supported by μS3D. The platform comes equipped with a preloaded sample 3D model. Through the web application, it is possible to try the following 3D manipulation functionalities:

1) Visualize the 3D model and explore its various assemblies.
2) Control the visibility of different assemblies, choosing which ones to show or hide.
3) Modify the color of any assembly.
4) Apply textures to specific parts by mapping images onto them.
5) Create a personalized avatar based on the model.
6) Customize the animation of the model.
7) Easily obtain a download link for the customized model or the HTML code required to embed it within a web application.

Figure 4 shows an example of platform usage, where some elements have been selected and others have not, and where the textures for some assemblies has been changed either by applying a color or mapping an image. In this example, we have illustrated how, by leveraging these functionalities, prospective developers could seamlessly create the functionalities that permit to interact with the 3D models, tailor them, and conveniently integrate them into their applications that support web technology.

5 *μS3D* Evaluation: Monolithic versus Microservices Architecture

In this section we evaluate the effectiveness and performance of *μS3D*. We first describe the research questions we are seeking to investigate.

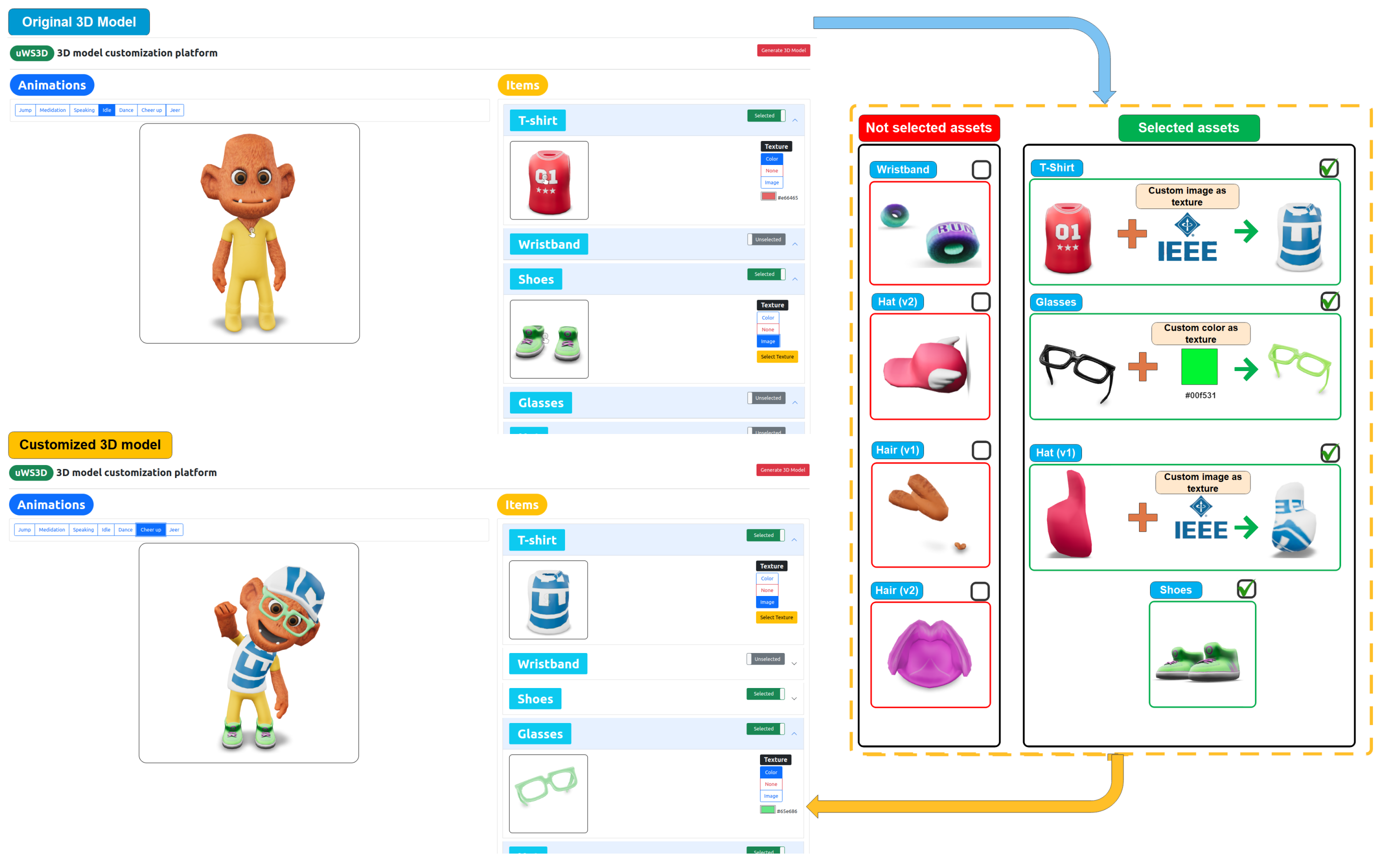


Fig. 4: Illustrative example of 3D Customization example in*µS3D* through a synthetic web platform

5.1 Research Questions

We designed our experimental study to answer the following research questions (RQs):

- **Q1:** How does our solution perform when compared to the monolithic implementation, which might be preferred by some developers?
- **Q2:** How would our microservices-based solution perform as the number of users increases?
- **Q3:** In terms of software development or software engineering, is a microservice-based solution always the best approach regardless of the domain, level of expertise or level of usage (number of users, traffic)?

5.2 Preliminary research for RQs

Analyzing the RQs, and before to perform the evaluation of our proposal, two different concern must be addressed: *1)* What would be the alternative to a microservices architecture in order to compare our solution (Q1) and *2)* what are the metrics required to measure the performance of our solution (Q1 and Q2).

After a comprehensive literature review, most surveys present the monolithic architecture as the the alternative to a microservices architecture, which is the type of architecture that was commonly used prior to the emergence of microservices and remains highly relevant today [29], [30].

On the other hand, the first relevant metric in any microservice-based solution is the response time in the consumption of the service, that is, the time it takes for a system or application to respond to a request or perform a task (also known as latency) [31]. However, since microservice and monolithic architectures are supported by different technologies, efficient hardware usage is also a relevant metric to compare both solutions, particularly when considering scalability issues [32].

Thus, we will consider monolithic architecture as an alternative implementation to our microservice-based implementation and we will measure the *1)* response time, *2)* average CPU use and *3)* average memory use as hardware usage in our experiments.

A *µS3D* fork has been re-coded in a second implementation following a monolithic architecture approach for comparison purposes. The monolithic architecture implementation is subsequently described in the next subsection.

Thereafter, we describe the methods for the conducted experiments, the considerations that have been made and technical information about thedatasets obtained. The experimental results as well as the discussion about them are described in Section 6.

5.3 Monolithic Architecture Implementation

A monolithic architecture presents a single-server application where the front-end, data and functionality are included in a single platform [33], [34]. Monolithic applications have been (and still are) widely used by developers, companies and research centers, mainly for three main reasons: simplicity of design which allows for a comprehensive understanding of the entire system and facilitates the design and refactoring of different components; simplicity of development in that most development tools and integrated development environments (IDEs) are still geared towards developing monolithic applications; and finally, simplicity of deployment since most IDEs include a dedicated set of functionalities for deploying applications both locally and to external environments [34].

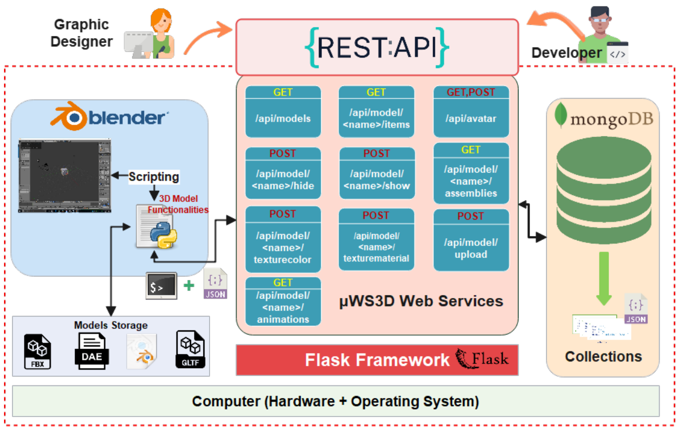

Graphic Designer
{REST:API}
Developer
blender
Scripting
3D Model Functionalities
GET /api/models
GET /api/model/<name>/items
GET,POST /api/avatar
POST /api/model/<name>/hide
POST /api/model/<name>/show
GET /api/model/<name>/assemblies
POST /api/model/<name>/texturecolor
POST /api/model/<name>/texturematerial
POST /api/model/upload
GET /api/model/<name>/animations
μWS3D Web Services
mongoDB
JSON
Collections
Models Storage
FBX
DAE
GLTF
Flask Framework
Flask
Computer (Hardware + Operating System)

Fig. 5: *μS3D* Monolithic Implementation

This implementation of the *μS3D* platform according to a monolithic design is supported by a service-oriented architecture (SOA), as depicted in figure 5 [35]. As figure 5 shows, the entire system runs on a single computer and implements the same example as the one presented in Figure 3. The system is composed by a set web-services deployed in a single server supported by Flask, a web framework written in Python[8]. On top of Flask, different web services have been deployed for the customization of 3D models and to be consumed by end-users (e.g., graphic designers, developers) through a REST API. The different web services access a MongoDB database to store 3D fingerprint models, and interact with a Blender engine through a scripting software implemented in Python to customize the 3D models stored in the persistent storage. A custom HTML-based visualizer with the model is returned to the end-user when 3D model has been customized.

The various services described in Section 3 have been implemented as web services in Python and Flask. The computer graphics tool Blender has been used to generate customized 3D assets and the *wrapper* remains the same as the one used in the microservices-based implementation.

MongoDB [9] was chosen as the database management system to store the 3D model fingerprints since it is NoSQL and stores the information as JSON documents, the same notation used for 3D model fingerprints. In this way, CRUD (Create, Read, Update, Delete) operations relating to 3D model fingerprints and schemes are natively supported [36].

Unlike the microservices architecture, monolithic architectures present several drawbacks that become increasingly relevant, especially as the application grows in size. One of the most important drawbacks is that it is extremely difficult to scale the application [34]. In fact, anything concerning change is a disadvantage in a monolithic-based application [37]. Most of these drawbacks (particularly the issue of scalability) are overcome in the microservice implementation. The following section explores these differences in terms of performance through several load-tests conducted in both implementations.

8. https://flask.palletsprojects.com/
9. https://www.mongodb.com/

5.4 Methods

With the aim to check the performance between both implementations, three different metrics have been collected: *1)* time consumption, *2)* average CPU use and *3)* average memory use.

We have simulated a set of invocations from end-user applications (where each runs a set of HTTP requests to consume the different functionalities described in Section 3) against the microservices-based and monolithic architectures. The same number of end-user applications (and requests) have been tested in both implementations. The following considerations have been made:

1) Each user (requester) runs a total of eight requests in a row to the REST API (HTTP). All the users run the same number of requests in the same order.
2) The requests are: Obtain the models, Obtain the model animations, Obtain the model parts, Hide a model part, Show a model part, Change the texture (map) for a color, Change the texture (map) for an image, and, finally, Generate the 3D model.
3) All users have run the requests with the same 3D model.
4) Both implementations have been tested with *1, 3, 5, 10, 25, 75, 100* and *200* users (*that suppose 8, 24, 40, 80, 200, 400, 800* and *1600* requests, respectively).
5) In each experiment (1,3,5,10,25,75, 100 and 200 users), all users runs in parallel with the aim to simulate a real scenario where a group of users attempt to customize the same 3D model.
6) The cache-based system supported by MongoDB for 3D model fingerprints in both implementations has been disabled. Using the cache-based system, the metrics since the second request remain the same, defeating the purpose of the test.
7) Each *μS3D* implementation (monolithic, microservices) are run on separate virtual machines[10] with the same features: 2 core processors, 8GB RAM, 50GB HDD and Ubuntu 20.04 LTS (Focal Fossa) as the operating system.
8) Both virtual machines run on a host computer with the following specifications: Intel® CoreTM i7-8700K CPU 3.70GHz, 64GB RAM DDR4 and 2TB SSD with Ubuntu 21.10 Impish Indri as the operating system.
9) Both implementations have been tested at different moments in time in order to ensure each implementation has the same available resources provided by the host computer.
10) Due to Blender's high system requirements, this has been limited to a maximum of three simultaneous

10. https://www.virtualbox.org/

process instances in both virtual machines and this is managed by the operating system.

11) Both the test plans (the series of requests to be executed) and the metrics of the various tests have been performed using *Apache JMeter*[11].
12) Both the test plans and the output metrics(time, average CPU and memory use) of tests of both implementations are available from the repository described in the *Supplementary Material* section.
13) To measure resource utilization (CPU, Memory) we have used a cross-platform lib for process and system monitoring[12].
14) Resource utilization can be directly collected in the monolithic implementation. However, the microservice-based implementation uses Docker containers, which are designed to work in isolation, so the access to the server where all microservices are running it is not possible. Therefore, for this experiment, we have developed a dedicated ad-hoc service that records for each operation its timestamp to gather resource utilization information. This service has been deployed to be consumed by the rest of microservices.

Once we conduct the tests, we obtained the entire dataset, which is available in the repository found in the *Supplementary Material* section.The details regarding the dataset are the following:

1) The dataset is a set of files represented in CSV (Comma-separated Values) format. There are a set of files for metrics related to *1)* the time consumption and another set of files for *2)* resource utilization (CPU and memory).
2) For time consumption, there are two folders, one per each implementation (i.e., monolithic and microservice-based), bot located under the path *JMeterDataset/TimeConsumption*. Each implementation folder contains a set of folders named according to the number of users (1,3,...,200), and which contains the six files generated by Apache JMeter. The only one used for this studio has been the file *ws3d_<implementation>_<numberusers>_users_table.csv*. In this file, the relevant columns are the timestamp, the elapsed time and the label, which contains the number of operation and the user that makes each request (i.e., *Op-7-ChangeTextureMaterial-76* represents that the user #76 mapped a texture to the 3D object).
3) As for resource utilization, there is a folder named *mono* and another one named *micro* under a root folder named *ResourceUtilization*. Inside these folders there are a set of files named according to the number of users (*mono_1.csv, mono_50.csv, micro_200.csv*). Relevant columns are *timestamp*, *id* for the user number, *operation* for the operation performed (i.e., get models, change texture, etc.), *totalcpuusagepsutils* for the total utilization of CPU and *memoryusage* for the memory utilization.

11. https://jmeter.apache.org/
12. https://pypi.org/project/psutil/

6 Load-Testing Results

6.1 Evaluation of Time response

Regarding the first metric, i.e., time response, Figure 6a shows a comparative overview of the results of the two experiments and displays the number of requests, time consumption and the different experiments per number of users in both implementations.

Both implementations obtain similar results for a moderate number of users. However, when there is a significant increase in the number of users and therefore the number of HTTP requests, the microservices implementation shows a better time consumption and therefore performance compared to the traditional monolithic architecture. As expected, the test with 200 users running simultaneously is the one which requires the most time in both implementations (around 44 minutes in the microservices architecture against the 132 minutes required by the monolithic architecture).

On the other hand, Figure 6b shows a box plot to illustrate the statistical representation of the distribution of the time consumed by each type of service and microservice in both architecture implementations by 200 users running in parallel.

The time required for each type of service and microservice is significantly higher in the monolithic architecture except in the *Show complement*, where the time required is a little big higher in the microservices architecture. However, in overall terms, a higher time consumption is needed in most services implemented in the monolithic architecture and this increases the time required to complete the run with 200 users.

Our preliminary results suggest that the microservices architecture works much better with a high number of users/requests. Unfortunately, we have not been able to carry out the test with more than 200 users due to limitational resources (host computer specifications).

6.2 Evaluation of hardware, CPU and memory, utilization

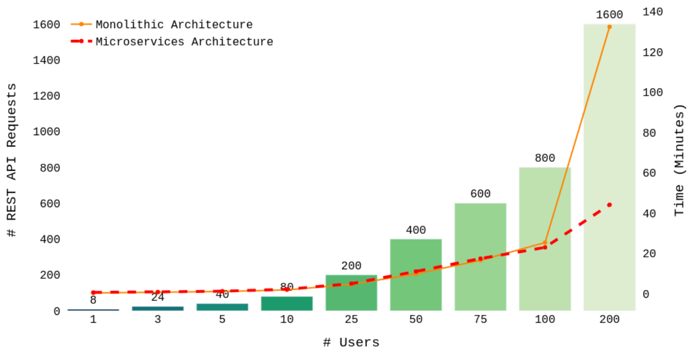
Concerning average CPU use, Figure 7a shows the results obtained in both implementations.

According to this results, CPU usage is lower in the microservices architecture. In the most demanding situation, with 200 users, the CPU usage reached 99.36% in the monolithic implementation, while in the microservice implementation, that value was 87.41%. The microservice implementation demonstrates higher efficiency, especially for a smaller number of users, and exhibits more stable performance as the number of users increases compared to the monolithic architecture.

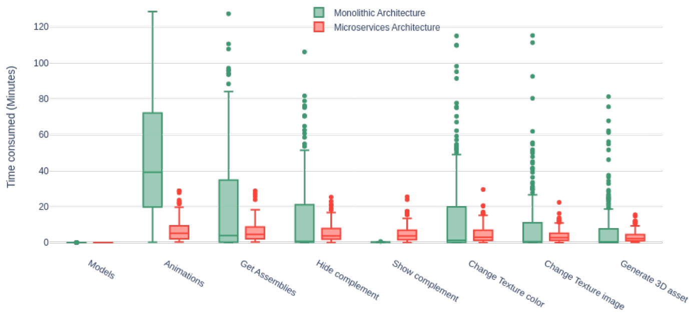
Otherwise, the memory usage remains more or less constant in both implementations, regardless the number of users, but it is clearly higher, and therefore, with poorer performance, in the microservices-based architecture (around 20% for monolithic implementation and around 50% for microservice-based implementation) (see Figure 7b).

6.3 Estimated response time and use of hardware resources with a higher number of users

In order to improve the performance of the microservices architecture implementation, we have used a forecasting

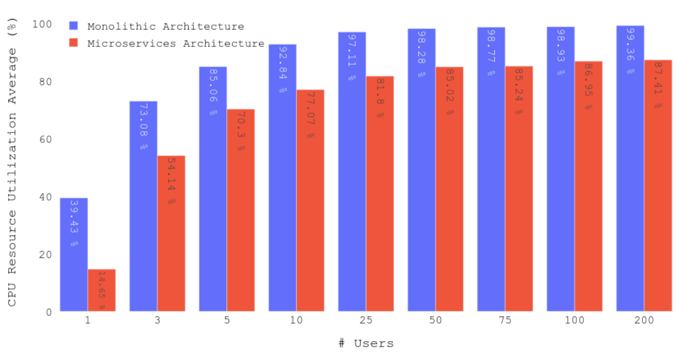


(a) Time consumption in monolithic versus microservices implementations

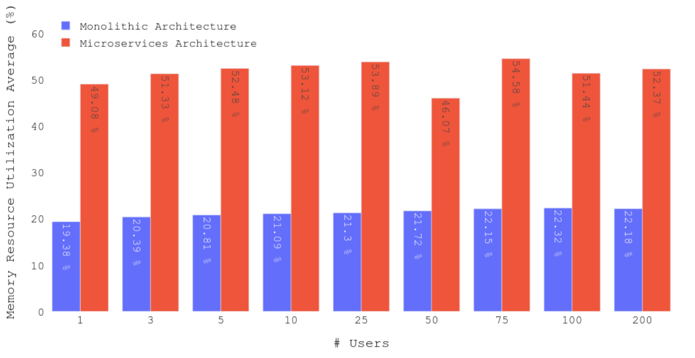


(b) Time consumption distribution for 200 users running in parallel

Fig. 6: Time consumption comparison



(a) Average CPU use



(b) Average Memory use

Fig. 7: Average CPU and memory use comparison (%)- Monolithic vs Microservices Implementations

approach to make predictions for a higher number of users using the data gathered from the test with 200 users for both implementations. We have used an *AutoregRessive Integrated Moving Average (ARIMA)* model [38] to predict future time consumption as well as CPU and average memory use. Like many other researchers, we have conducted a train-test split procedure using the classical 70-30 split (70% of samples for training and 30% for testing 140 and 60 users, respectively). ARIMA trains the model with the input dataset (train) to find the linear regression and make a prediction over the test dataset. Figure 8a shows the prediction for test dataset of monolithic implementation and Figure 8b illustrates the prediction for microservices implementation for time consumption metric. The CPU usage values have a maximum limit (100%) that is almost reached during the conducted tests, while the memory usage remains constant in both implementations ((see Figure 8c and 8d for CPU usage and Figure 8e and 8f for memory average use).

With both the predicted results and the original values (test dataset), we have used a root-mean-square error (RMSE) method to compare the difference between both sets of values [39] in order to quantify model confidence and reliability. The RMSE for the different metrics are as well as normalized (NRMSE) can be shown in Table 2.

A zero value as the estimated error indicates that the model is perfect. As the values (minutes) are in the order of tens (microservices) and hundreds (monolithic), the estimated error value implies that the prediction fits well with the real values, and the model trained for the microservices implementation is more reliable. On the other hand, although NRMSE are better for CPU than for average memory use, making them more reliable, in both cases, the values are not far from 0, so it can be concluded that both trained models are reliable.

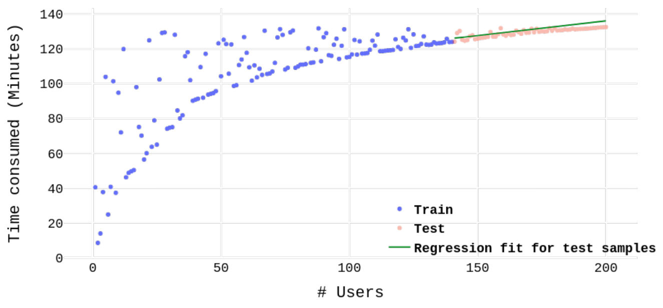
TABLE 2: Summary of the RMSE and NRMSE values for the three metrics

	Monolithic			**Microservice**		
	Time	*CPU*	*Memory*	*Time*	*CPU*	*Memory*
RMSE	2.19	4.18	0.949	1.09	4.63	0.99
NRMSE	0.1768	0.04	0.254	0.1752	0.05	0.23

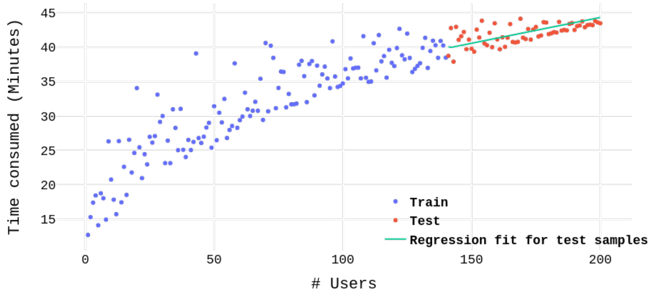
As we are certain that our predictions are quite similar to real values, we have again applied the same trained model to predict the time consumed by the tests conducted for 400, 600, 800 and 1000 users. Table 3 summarizes the time consumption of both implementations for the tests conducted as well as the predicted values.

The three models have been trained with the following ARIMA features:

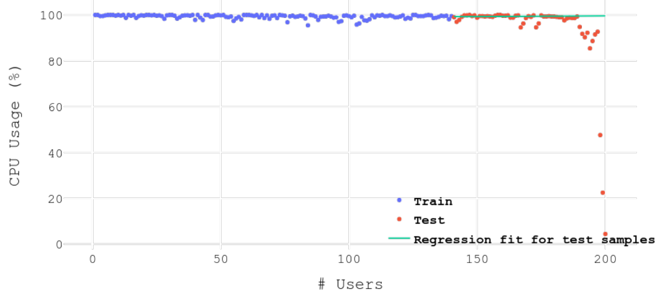
1) 2 as the number of lag observations or autoregressive terms in the model or *AR (p)* ;



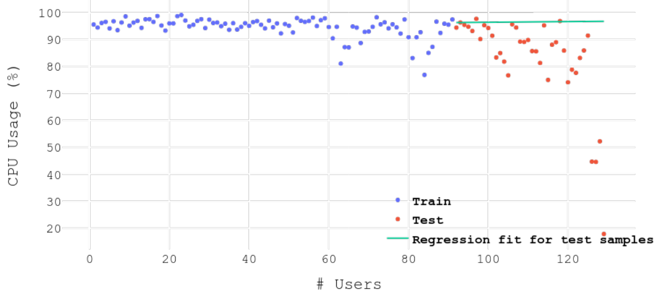
(a) Time Consumption - Monolithic



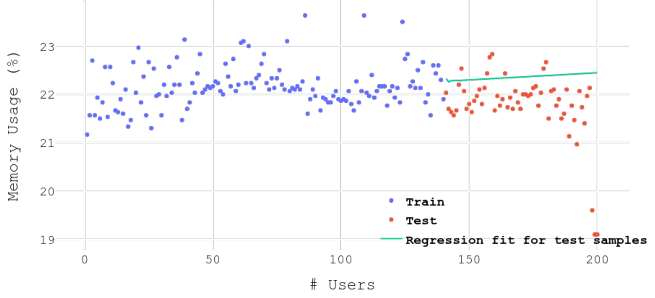
(b) Time Consumption - Microservices



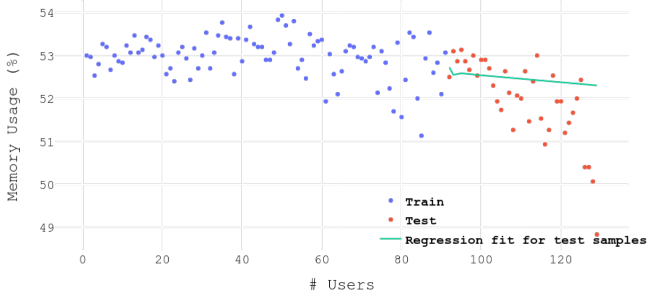
(c) CPU Average Use - Monolithic



(d) CPU Average Use - Microservice



(e) Memory Average Use - Monolithic



(f) Memory Average Use -Microservice

Fig. 8: Forecasting time consumption as well as CPU and memory average usage through ARIMA model to customize 3D models

2) 2 as the difference in the nonseasonal observations *I (d)*;
3) 2 the size of the moving average window or *MA (q)*.

In addition, for the forecasting, alpha is 0.05, which means that the ARIMA model will estimate the upper and lower values around the forecast where there is only a 5% chance that the real value will not be in that range.

6.4 Evaluation of RQs

The conclusions that can be drawn regarding the RQs are as follows.

Evaluation of Q1

How does our solution perform when compared with most widely adopted implementation other than microservices?

- Irrespective of the 3D model part managed by the developer or 3D designer (textures, animations, assemblies, etc.), the microservices architecture provides a better response time and better utilization of the CPU.
- Memory usage does not change too much regardless of number of users. So, once the minimum (or recommended) memory requirements of the 3D computer graphic tool are met, additional memory does not improve performance in any aspect.
- The average memory usage is constant in both implementations, but significantly higher in the microservices architecture (twice as much). This is be-

TABLE 3: Comparison in both implementations for the time consumption and resource utilization

# Users	Monolithic						Microservices					
	Time Consumption (minutes)		CPU Usage (% avg)		Memory Usage (% avg)		Time Consumption (minutes)		CPU Usage (% avg)		Memory Usage (% avg)	
	Test	*Predicted*	*Test*	*Predicted*	*Test*	*Predicted*	*Test*	*Predicted*	*Test*	*Predicted*	*Test*	*Predicted*
1	0.31		39.43		19.38		0.59		14.65		49.08	
3	0.65		73.08		20.39		0.82		54.14		51.33	
5	1.01		85.06		20.81		1.21		70.3		52.48	
10	1.84		92.84		21.09		2.17		77.07		52.12	
25	4.59		97.11		21.3		4.98		81.8		53.89	
50	9.81		98.28		21.72		11.01		85.02		46.07	
75	16.55		98.77		22.15		17.41		85.24		54.58	
100	25.36		98.93		22.32		22.98		86.95		51.44	
200	132.55	136.12	99.36	99.39	22.18	22.32	44.14	44.34	87.41	96.22	52.37	52.72
400		159.81		100		22.85		54.83		98.52		50.97
600		193.66		100		23.44		69.82		100		49.32
800		227.51		100		24.03		84.81		100		47.68
1000		261.19		100		24.62		99.72		100		46.04

cause microservice implementation needs a required technology stack, i.e., Docker and Kubernetes, which inherently consumes memory even when not actively processing requests, regardless of the number of users.

Evaluation of Q2

How would our microservices-based solution perform with a higher number of users in terms of performance?

- The microservices architecture performs better with a large number of users as far as time consumption and average CPU use are concerned.
- For a reduced number of users or testing purposes, monolithic implementation is more efficient.

Evaluation of Q3

In terms of software development or software engineering, it's a microservice-based solution always the best approach regardless of the domain, level expertise or level of usage (number of users, traffic)?

- The monolithic architecture implementation is simpler than the microservices architecture in terms of development and deployment, so the monolithic architecture approach is probably much more suitable for small-medium companies or independent developers/programmers.
- The deployment of a microservice-based architecture demands specialised knowledge in Kubernetes and Docker, so an expert programmer it is required.

7 Discussion

The solution presented in this study was designed to support a fully customization of 3D models without any technological background about 3D modeling tools by developers. The different contributions of the work have outstanding benefits:

- µS3D is a technology that makes the generation of custom 3D models transparent for developers, enabling the integration process of 3D models into any kind of platform.
- µS3D technology eliminates the need for 3D designers in the modification phases.
- The 3D model fingerprint included as part of the proposal allows for model generation optimization by reusing previously generated models, reducing response times and storage requirements.
- Both implementations, monolithic and microservices-based exhibit reasonable response times, but the microservices-based one has shown better performance values.
- The *wrapper* has been designed to work with Blender. However, new wrappers can be added to the platform so as to support additional third-parties 3D manipulating engines, such as 3DStudio Max.
- uS3D provides customized 3D models in a standard format such as GLTF. Additionally, it offers an HTML-based visualizer that allows seamless integration of the model into any platform.

Conversely, the proposal presented still has some challenges and limitations.

First of all, current version of µS3D has been designed to customize the 3D models/assets both before and after the animation scene. For instance, we are able to customize assemblies and textures before the start of the animation (i.e., dance, jump, walk) or at the end of the animation. However, the technology introduced does not support customization during the transition of a dynamic 3D scene. In other words, it is not possible to change the texture of the model or an assembly in the middle of an animation.

In second place, µS3D eliminates the need for a designer to customize the 3D model once it is finished, including all assemblies and animations. However, if new assemblies or animations need to be added to the 3D model, the designer's participation is still required. One limitation of our proposal is the absence of services that enable the automatic merging of different 3D models. Such services would support the addition of new assemblies or animations from the applications that manipulate them.

Finally, although µS3D enhances the integration of 3D models in any platform by developers/programmers without any 3D technology background, this is for developers who use µS3D once it is deployed and running. However, the deployment of µS3D, could be challenging. The monolithic implementation is quite easy because it runs as any other back-end software (i.e., with a single command it is possible to run the platform), but the microservices-based implementation requires knowledge about Kubernetes, Dockers and administration tasks to run the entire infrastructure. In this way, although the quick start guide presented in the *Supplementary Materials* section is provided as assistance, at least one developer with a medium-high background about microservices would be required.

These disadvantages or shortcomings are considered manageable and will be addressed as future work.

8 Conclusions and Further Work

3D assets are usually modeled by graphic designers to provide the developers with models (files) so that they can be integrated into software applications, e.g. 3D engines, libraries or software development kits (SDKs). Although this might appear to be a fairly simple procedure, developers must deal not only with the lack of knowledge regarding 3D computer graphics and the use of dedicated technologies, objects representations and formats, but also the continual communication with designers in case the 3D assets need to be modified or customized. Generally speaking, this scenario requires the application to be rebuilt and this will affect the end-user experience as new app downloads or updates are required.

In this paper, we present *µS3D*, a microservice platform for customizing 3D model elements (e.g. armatures/skeletons, animations, textures) which comprises three core components: a *microservices platform* consisting of a series of microservices, a *wrapper* to interact with an underlying computer graphics engine, and the *3D model fingerprint*, a cache-based system to enhance performance in terms of storage and response time in the generation of customized 3D models.

µS3D is able to overcome classical procedure issues and significantly enhance the integration of 3D assets into apps since firstly, it enables 3D assets to be customized without the need for graphic designers simply by orchestrating different microservices, secondly, it does not require any previous knowledge or experience of 3D computer graphics or hard programming skills, and thirdly, it allows 3D assets to be displayed directly online without the need to download the corresponding models. Outsourcing of 3D assets management functionalities also contributes to lighten the size of software applications that make use of them.

Two separate implementations following a microservice-oriented approach and a monolithic approach respectively, have been developed in order to illustrate the feasibility of the proposal. Furthermore, various load and stress tests have been conducted to simulate a number of invocations in the microservice platform as well as in the µS3D fork implemented following a monolithic approach. Our tests show that performance is remarkably better in the microservices architecture implementation, especially when the system is used by a large number of users. Conversely, the monolithic implementation also performs well with a reduced number of concurrent users.

By way of future work, we intend to explore a number of different research avenues. First, we plan to collaborate with graphic design experts to support new features through additional microservices in order to servitize functionalities which are currently only implemented using computer graphic tools (i.e. real-time modeling/animation, rigging).

The *µS3D* microservices platform is currently supported by a cluster with a single node. In order to improve this implementation, we also intend to implement an improved microservices platform which is supported by a distributed architecture (a cluster with various independent hardware nodes) in order to replicate the conducted load-tests and to compare the performance of the three different implementations (monolithic, centralized microservices and distributed microservices).

In this line we plan to deploy it in a real environment with sufficient available resources to validate our comparison like a cloud computing platform (Microsoft Azure®, Google Cloud®, or Amazon Web Services®) and using different 3D models instead a single case study like the one presented in this paper for the comparison.

Although the metrics obtained through the conducted load-testing experiments are very promising, additional tests are required to validate them. For instance, it is necessary to perform a set of formal experiments to check the accuracy of our prediction model or even compare the results by applying different algorithms.

Finally, the shortcomings mentioned in the *Discussion* (support customization during 3D scene animations and enable the combination of models, thus avoiding the need for a 3D designer to extend a model with new assemblies or animations) (see Section 7) will be addressed to be supported in the next version of the platform.

Acknowledgments

This paper was funded by the Spanish Ministry of Science and Innovation under R&D&I projects GOMINOLA (PID2020-118112RB-C21 and PID2020-118112RB-C22); by the ”Plan Andaluz de Investigación, Desarrollo e Innovación” (PAIDI 2020) and, finally, by Proyectos de investigación orientados a los retos de la sociedad andaluza” under contract P18-RT-4550 (BonAppPetit project).

Supplementary Materials

Supplementary materials associated with this article are available at https://github.com/bihut/microws3sd A video demonstration of how µS3D works can be found at https://www.youtube.com/watch?v=Q8GZWp_Szgc

REFERENCES

[1] F. Seron, J. Martin-Albo, A. Zaldivar, J. Magallon, and A. Blesa, "A semantic memory bank assisted by an embodied conversational agents for mobile devices," *Engineering and Applied Sciences*, vol. 6, pp. 1–17, 2021.

[2] J. Reinhardt, L. Hillen, and K. Wolf, "Embedding conversational agents into ar: Invisible or with a realistic human body?" in *Proceedings of the Fourteenth International Conference on Tangible, Embedded, and Embodied Interaction*, 2020, pp. 299–310.

[3] W. Huang, R. D. Roscoe, M. C. Johnson-Glenberg, and S. D. Craig, "Motivation, engagement, and performance across multiple virtual reality sessions and levels of immersion," *Journal of Computer Assisted Learning*, vol. 37, no. 3, pp. 745–758, 2021.

[4] M. Podroužek, J. Matišák, and K. Žáková, "3d model of the thermo-opto-mechanical plant for use in control education," in *2020 Cybernetics & Informatics (K&I)*. IEEE, 2020, pp. 1–5.

[5] P. Bitrián, I. Buil, and S. Catalán, "Enhancing user engagement: The role of gamification in mobile apps," *Journal of Business Research*, vol. 132, pp. 170–185, 2021.

[6] M. L. Brutto and P. Meli, "Computer vision tools for 3d modelling in archaeology," *International Journal of Heritage in the Digital Era*, vol. 1, no. 1_suppl, pp. 1–6, 2012.

[7] A. Voinea, F. Moldoveanu, and A. Moldoveanu, "3d model generation of human musculoskeletal system based on image processing: An intermediary step while developing a learning solution using virtual and augmented reality," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, 2017, pp. 263–270.

[8] K. H. Sharif and S. Y. Ameen, "Game engines evaluation for serious game development in education," in *2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2021, pp. 1–6.

[9] C. Vohera, H. Chheda, D. Chouhan, A. Desai, and V. Jain, "Game engine architecture and comparative study of different game engines," in *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, 2021, pp. 1–6.

[10] A. Scianna, "Building 3d gis data models using open source software," *Applied Geomatics*, vol. 5, pp. 119–132, 2013.

[11] J. Robles, C. Martín, and M. Díaz, "Opentwins: An open-source framework for the design, development and integration of effective 3d-iot-ai-powered digital twins," 2023.

[12] E. Al-Masri, "Enhancing the microservices architecture for the internet of things," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 5119–5125.

[13] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in devops," *Journal of Systems and Software*, vol. 170, p. 110798, 2020.

[14] J. Sanchez-Riera, A. Civit, M. Altarriba, and F. Moreno-Noguer, "Avatar: Blender add-on for fast creation of 3d human models," *arXiv preprint arXiv:2103.14507*, 2021.

[15] L. Briceno and G. Paul, "Makehuman: a review of the modelling framework," in *Congress of the International Ergonomics Association*. Springer, 2018, pp. 224–232.

[16] K. Walczak and J. Flotyński, "Inference-based creation of synthetic 3d content with ontologies," *Multimedia Tools and Applications*, vol. 78, no. 9, pp. 12 607–12 638, 2019.

[17] J. Flotyński, K. Walczak, and M. Krzyszkowski, "Composing customized web 3d animations with semantic queries," *Graphical Models*, vol. 107, p. 101052, 2020.

[18] Y. Yoon, K. Park, M. Jang, J. Kim, and G. Lee, "Sgtoolkit: An interactive gesture authoring toolkit for embodied conversational agents," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 826–840.

[19] X. Piao, Y. Li, K. Xie, H. Zhao, and J. Jia, "Towards web3d-based lightweight crowd evacuation simulation," in *The 25th International Conference on 3D Web Technology*, 2020, pp. 1–9.

[20] Z. Liu, L. Li, F. Tian, and J. Jia, "Lightweight web3d crowd rendering for online massive conferencing," in *2022 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. IEEE, 2022, pp. 536–541.

[21] P. Nicolaescu, G. Toubekis, and R. Klamma, "A microservice approach for near real-time collaborative 3d objects annotation on the web," in *Advances in Web-Based Learning–ICWL 2015: 14th International Conference, Guangzhou, China, November 5-8, 2015, Proceedings 14*. Springer, 2015, pp. 187–196.

[22] F. Hong, M. Zhang, L. Pan, Z. Cai, L. Yang, and Z. Liu, "Avatarclip: Zero-shot text-driven generation and animation of 3d avatars," *arXiv preprint arXiv:2205.08535*, 2022.

[23] T. Wang, B. Zhang, T. Zhang, S. Gu, J. Bao, T. Baltrusaitis, J. Shen, D. Chen, F. Wen, Q. Chen *et al.*, "Rodin: A generative model for sculpting 3d digital avatars using diffusion," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 4563–4573.

[24] T. Nguyen-Phuoc, G. Schwartz, Y. Ye, S. Lombardi, and L. Xiao, "Alteredavatar: Stylizing dynamic 3d avatars with fast style adaptation," *arXiv preprint arXiv:2305.19245*, 2023.

[25] C. Zhang, Y. Chen, Y. Fu, Z. Zhou, G. Yu, B. Wang, B. Fu, T. Chen, G. Lin, and C. Shen, "Styleavatar3d: Leveraging image-text diffusion models for high-fidelity 3d avatar generation," *arXiv preprint arXiv:2305.19012*, 2023.

[26] G. Llorach, J. Agenjo, J. Blat, and S. Sayago, "Web-based embodied conversational agents and older people," in *Perspectives on Human-Computer Interaction Research with Older People*. Springer, 2019, pp. 119–135.

[27] P. Szalaj, P. J. Michalski, P. Wróblewski, Z. Tang, M. Kadlof, G. Mazzocco, Y. Ruan, and D. Plewczynski, "3d-gnome: an integrated web service for structural modeling of the 3d genome," *Nucleic acids research*, vol. 44, no. W1, pp. W288–W293, 2016.

[28] M. Wiederstein and M. J. Sippl, "Prosa-web: interactive web service for the recognition of errors in three-dimensional structures of proteins," *Nucleic acids research*, vol. 35, no. suppl_2, pp. W407–W410, 2007.

[29] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," *Applied sciences*, vol. 10, no. 17, p. 5797, 2020.

[30] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.

[31] J. Correia, F. Ribeiro, R. Filipe, F. Arauio, and J. Cardoso, "Response time characterization of microservice-based systems," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–5.

[32] A. Akbulut and H. G. Perros, "Performance analysis of microservice design patterns," *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019.

[33] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 93–96.

[34] N. Lamouchi, "Introduction to the monolithic architecture," in *Pro Java Microservices with Quarkus and Kubernetes*. Springer, 2021, pp. 35–38.

[35] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari *et al.*, "Understanding service-oriented architecture (soa): A systematic literature review and directions for further investigation," *Information Systems*, vol. 91, p. 101491, 2020.

[36] T. N. Khasawneh, M. H. AL-Sahlee, and A. A. Safia, "Sql, newsql, and nosql databases: A comparative survey," in *2020 11th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2020, pp. 013–021.

[37] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[38] J. Fattah, L. Ezzine, Z. Aman, H. El Moussami, and A. Lachhab, "Forecasting of demand using arima model," *International Journal of Engineering Business Management*, vol. 10, p. 1847979018808673, 2018.

[39] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.