Yoop – Serverless Functions Platform

RARES CHIUZBAIAN
Email: chiuzbaian.vi.rares@student.utcluj.ro

CATALIN IUGA
Email: iuga.gh.gheorghe@student.utcluj.ro

NORBERT KOVACS
Email: kovacs.al.norbert@student.utcluj.ro

GABRIEL MURESAN
Email: muresan.cl.gabriel@student.utcluj.ro

MADALINA PODE
Email: pode.ga.madalina@student.utcluj.ro

EDUARD PODOLAC
Email: podolac.ad.eduard@student.utcluj.ro

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA
Faculty of Sciences
Specialization: Software Engineering

Abstract

Yoop is a cloud platform for developing serverless functions with a focus on simplicity, security, and low cost. It allows businesses and individuals to pay only for the resources they use, and scales functions based on usage and available processing power. Yoop also offers analytics to deep dive into function performance and usage, while exploring predictive solutions for cost estimation and performance improvements.

Keywords: *distributed systems, serverless, cloud-computing*

Introduction

Suppose you have developed a cool web application, but your friends are unable to access it. In such a scenario, you will need to find a hosting platform, get a server or virtual machine, obtain a domain, learn about Docker, CICD, and automated deployments, and pay for the big bill since your machines will be running continuously. Hosting an API can be a daunting task, given the numerous options available, hundreds of providers, different deployment types, and how quickly costs can accumulate. This is where serverless functions come in. They run your code on-demand, saving resources for the provider, which translates into lower costs for the end-user.

This is one of the reasons why we decided to a serverless function platform. Our platform, Yoop, aims to simplify the process of developing serverless functions and deploying them on a cloud infrastructure. With Yoop, users can easily write functions without worrying about the hassle of setting up automated workflows for deployment. Our goal is to provide a user-friendly and efficient platform that enables people to create and deploy serverless functions effortlessly.

The existing cloud solutions tend to get expensive quickly. Big companies like Amazon, Google or Microsoft offer a large range of products and they're mostly targeted at corporations with sufficient financial resources that can be used to maintain building solutions using existing cloud products. What we're trying to do is offer a niched approach, focusing on serverless computing, offering simple, straightforward, and secure deployments.

Most cloud vendors allow the development of a function to be done by writing the code directly in a specific language and we're sticking to this approach, but we're also looking to build an interactive function builder which can allow non-technical people to logically build their own functions using building blocks for actions, conditional statements, loops and so on.

The main objectives of Yoop should be investing in a lower-cost infrastructure, as a business, and providing a lower-cost solution to building serverless computing solutions, for the users. The costs should be lower than any existing serverless computing solutions, allowing smaller businesses or individuals to pay low prices on computing units, used resources or function calling counts. The challenge would be to maintain an acceptable level of performance and security, while also providing accessible prices per function deployment.

We're building Yoop with scalability in mind, focusing on scaling user-built functions based on usage, and routing requests based on available processing power for each function instance.

The control and access to resources exhausted is also an important factor we're focusing on, and we've decided to build an analytics system which allows users to deep-dive into the performance of the functions, function usage and the costs involved in running these functions. Recent years marked a lot of innovations in the AI and ML fields, so we're looking into building predictive solutions to allow users to determine potential usage and costs of running their serverless functions, but also to suggest performance improvements and perhaps, bottlenecks.

Technologies

Developing a project of a large scale and complexity calls for a strategic approach towards choosing the right programming languages and frameworks. The decision primarily hinges on the nature of the project and the proficiency of the developer. At Yoop, we adopt a flexible approach by using different technologies that each member is comfortable with, based on the task at hand. This approach enables us to develop features and services faster without the need to spend time researching and learning the same type of programming language or framework.

This not only saves time but also leads to a more efficient and timely delivery of the project. Additionally, by allowing developers to work with the languages they are most comfortable with, we encourage a more productive and collaborative work environment, where each team member brings their unique experience and expertise to the table.

Astro

Astro simplifies web development by offering efficient reactivity, seamless integration with existing projects, and a component-based architecture for scalable UI development [1]. It is the chosen frontend framework for the Yoop platform, facilitating modular UI components and dynamic user interfaces. Astro's SSR capabilities enable fast-loading, SEO-friendly interfaces, and its integration with Vue.js enhances client-side interactivity, making it the perfect fit for Yoop's serverless function platform. With Astro, Yoop provides a user-friendly and efficient platform for developing and deploying serverless functions, empowering users to focus on their core business logic.

Vue

Vue.js is a JavaScript framework that is used to create user interfaces and single-page applications (SPAs) in a progressive manner [2]. We chose Vue as the frontend for the Yoop platform because of its simplicity, well-managed reactivity, and easy integration with other libraries or existing projects. Vue is known for its reactive system that allows for efficient and concise updates to the user interface, which is essential for delivering dynamic user experiences on Yoop.

Vue follows a component-based architecture that makes it easier to develop the interface in a modular and scalable way, which is very important for managing the diverse functionalities of our platform. Moreover, Vue's gentle learning curve helped our team make rapid progress in developing the frontend without compromising on quality or performance.

Streamlit

Streamlit is a Python framework tailored for streamlined development of data-centric web applications, emphasizing interactive visualizations and dashboards. With its intuitive API, developers can effortlessly integrate various charting libraries like Matplotlib, Plotly, and Altair to create dynamic visualizations [3].

Streamlit simplifies dashboard creation by offering easy-to-use components for layout management and interactive widgets, enabling rapid prototyping and deployment. Its focus on data visualization makes it a go-to choice for data scientists and analysts seeking to share insights and analysis through user-friendly web interfaces.

In this scenario, Streamlit serves as a versatile tool for monitoring essential metrics like requests per second and cost in our serverless cloud platform.

NestJS

Built on top of Express, NestJS is an opinionated framework used for building efficient, reliable, and scalable server-side applications. One of the core features of NestJS is its modular architecture which allows developers to organize their code using modules. This helps in creating a loosely coupled and highly maintainable codebase.

NestJS is heavily inspired by Angular in terms of its architecture and design patterns, leveraging decorators for various class-based operations such as defining controllers, services, and modules. Some design patterns could lead you towards comparing it to Java Spring, in terms of the decorators used (@Controller, @Injectable, @Middleware, etc.).

NestJS is a fantastic backend framework which allows us to build various services efficiently, in an organized manner. We managed to quickly build the function manager and expose various methods to efficiently store and manipulate function configurations.

gRPC

gRPC is a modern, high-performance, open-source Remote Procedure Call (RPC) framework developed by Google. It allows a client application to directly call methods on a server application as if it were a local object, making it easier for developers to create distributed applications and services.

gRPC uses protocol buffers, which is an open-source mechanism for serializing structured data. This makes it more efficient than JSON, which in combination with HTTP/2 and its efficient use of network resources provides us a fast way of transferring data between the backend services.

The communication between the Proxy accepting requests and WebSocket connections from the browser and the function manager is made using gRPC. This allows efficient and secure communication between the Proxy, which is supposed to serve data to the client via HTTP and WebSocket, and the function manager.

Deno

Deno is a runtime environment for JavaScript and TypeScript that provides a secure and efficient platform for hosting user-supplied functions. Unlike Node.js, it addresses many of its shortcomings and offers better security features. One of its key features is that it runs each script in a secure sandbox by default, which means that it restricts access to sensitive operations like the file system and network, unless explicitly allowed [4].

This is one of the main reasons why we chose Deno as our hosting runtime for functions. We can fine-tune the permissions for each function, allowing only specific access to network calls and certain environment variables. This makes it impossible for malicious code to access the file system or any other part of our hosting environment.

Moreover, Deno's module system offers easy dependency management and code reuse, which simplifies the development and deployment process for our users. With Deno's native support for ECMAScript modules (ESM), developers can effortlessly import and use third-party libraries without the need for additional tools or configuration. We plan to take this into account as a future development perspective, allowing users to use 3rd party libraries as long as they match the specified permissions model.

Golang

Go, which was developed by Google, is a programming language that is statically typed and compiled [5]. It's widely recognized for its simplicity, efficiency, and ability to support concurrency.

It's an excellent choice for building scalable systems, which makes it highly suitable for our serverless function platform.

We have chosen Go as our primary language for managing the lifecycle of serverless functions within our platform and interacting with the Kubernetes client. This is due to its exceptional performance and reliability. Its support for concurrency through goroutines and channels enables us to efficiently handle multiple concurrent tasks, such as deploying and scaling functions, without compromising stability.

Go is a reliable choice due to its robust type system and efficient compilation process, which ensures dependable execution even in resource-constrained Kubernetes clusters. This reliability is essential for managing the deployment and lifecycle of serverless functions, where any downtime or instability can significantly impact application availability.

Moreover, Go's extensive standard library and active ecosystem of third-party packages offer comprehensive support for interacting with Kubernetes APIs and integrating with other systems such as RabbitMQ. This allows for seamless orchestration of serverless function deployment across distributed environments while ensuring reliable communication with other platform components.

Docker

Docker is a platform that simplifies the process of packaging, distributing, and running applications within containers. These containers encapsulate everything an application needs to run, including code, runtime, system tools, and libraries, ensuring consistency across different environments.

We have chosen Docker as a fundamental component of our serverless function platform because of its ability to streamline the deployment and management of functions in isolated environments. By containerizing each function, we can ensure consistent execution across various infrastructures, from local development environments to production clusters, while also providing isolation for the function processes.

Docker's robust ecosystem and tooling enable seamless integration with other technologies, like Kubernetes, allowing us to efficiently orchestrate the deployment and scaling of serverless functions across distributed environments. This ensures flexibility, scalability, and reliability, which are essential for a modern serverless platform.

.Net Core

.NET Core is an open-source, cross-platform framework developed by Microsoft for building distinct types of applications, including web, mobile, desktop, cloud, gaming, IoT, and more [6]. It is a modular framework that allows developers to leverage a consistent set of libraries and runtime across different platforms, such as Windows, macOS, and Linux.

We chose .NET Core because of its performance as it benefits from features such as just-in-time (JIT) compilation, ahead-of-time (AOT) compilation, and asynchronous programming support. These optimizations help improve the speed and responsiveness of your applications, making them more efficient and scalable. And because it is built with modularity in mind, allowing you to include only the necessary components in your applications. This modular approach helps reduce the overall footprint of your applications, improve performance, and streamline maintenance and updates.

Kubernetes

Kubernetes is an open-source container orchestration system, aimed at automating, scaling, and managing software deployment [7]. While it supports orchestrating multiple types of virtualized environments, we opted to use it alongside Docker as this combination best fits our needs.

We needed a way to automate the scaling and monitoring of the containers running the users' functions. At first, we thought of writing a monitoring tool ourselves that would use the Docker SDK to create new containers. We soon realized that this is one of the most used functionalities of Kubernetes and writing it from the ground up would bring no additional benefit. Consequently, Kubernetes also works as a load balancer between the different spawned containers associated with one function.

Ingress

In Kubernetes, Ingress helps expose and route HTTP traffic between the outside of the cluster and the services within it [8]. Ingress allows us to use name-based virtual hosting, fanout (different subdomain per function) and terminate SSL/ TLS.

For Ingress to work, an Ingress controller is needed. We chose Nginx Ingress as it is one of the most popular solutions, thorough documentation is available, and numerous community created blog posts, video tutorials and other resources are extremely easy to find.

Prometheus

Prometheus is a free and open-source application used for event monitoring and alerting. It works inside Kubernetes and processes metrics exposed by other such applications, like Nginx Ingress [9].

We required a method to collect data on the traffic and usage of each function, both for displaying this information to the creators of the functions and for our internal analysis. After settling on Kubernetes, we chose Prometheus as it is the industry standard for monitoring applications running in clusters.

Cassandra

Apache Cassandra is a free and open source, distributed, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. It was originally developed at Facebook and later released as an open-source project on Google Code. Cassandra offers robust support for clusters spanning multiple datacenters, with asynchronous masterless replication allowing low latency operations for all clients.

Unlike traditional database systems which use a master-slave architecture, Cassandra employs a peer-to-peer distributed system across its nodes, meaning there is no single point of failure. This architecture enhances its fault tolerance and data availability.

The function configurations are stored directly in Cassandra via the function manager. The NestJS service uses the Cassandra driver to establish a connection between a Cassandra instance and itself. We are using a custom keyspace called "yoop_platform" to store various tables, running on a single datacenter.

PostgreSQL

PostgreSQL is a powerful, open-source relational database management system (RDBMS) known for its reliability, robustness, and feature-rich capabilities. It is commonly used for storing and managing structured data in several types of applications, ranging from small-scale projects to large-scale enterprise systems.

We chose PostgreSQL for our relational database management system (RDBMS) because it is known for its reliability, stability, and robustness. It has a proven track record of being used in production environments for a wide range of applications, including high-traffic websites, enterprise systems, and critical infrastructure. And because PostgreSQL is fully compliant with SQL standards, offering a comprehensive implementation of SQL features and syntax. This ensures compatibility with existing SQL-based applications and tools and simplifies the migration of applications from other database systems.

General considerations

In the fast-evolving area of serverless computing, there are several platforms that offer unique approaches to function-as-a-service (FaaS) solutions. In this comparison, we assess our new serverless function platform, which is designed for both custom coding and block-based development, with established services such as ServiceNow Flows, AWS Lambda, and other serverless offerings.

ServiceNow Flows

ServiceNow Flows simplifies complex workflows using a visual programming interface like Scratch. This block-based approach is helpful for non-developers as it allows them to drag and drop components. However, it is primarily meant to work with other ServiceNow applications and is not very versatile for broader software development. Our platform, while still using a similar user-friendly interface for block-based function creation, extends functionality by enabling users to write custom code. This hybrid model facilitates not only workflow automation but also complex function deployment, catering to both novice and professional developers.

AWS Lambda

AWS Lambda is a vital part of modern serverless architectures, providing potent and scalable compute services without requiring the management of server infrastructure. Lambda supports a wide range of programming languages, excelling in handling high-demand applications. Unlike Lambda, which requires some understanding of cloud configurations and permissions, our platform simplifies the deployment process. We offer a more accessible entry point for beginners with templates and guides, while still supporting scalability and customization for advanced users.

Other serverless platforms

Other serverless platforms, such as Google Cloud Functions and Azure Functions, offer sturdy and scalable solutions but often require a steep learning curve related to cloud service configuration. These platforms, while versatile, can be intimidating for new users. Our platform sets itself apart by combining the ease of block-based programming with the power of custom code environments, making serverless computing accessible to a broader audience without sacrificing functionality.

While traditional serverless platforms like AWS Lambda offer extensive support for professional developers looking to build scalable applications, they can be less accessible for newcomers to the field. ServiceNow Flows offers an excellent solution for workflow automation within its ecosystem but does not cater as effectively to general software development. Our platform seeks to bridge these gaps, offering a user-friendly, flexible environment for both developing and deploying serverless functions. By allowing users to switch effortlessly between block-based and code-based development, we provide a unique tool that is adaptable to various user needs, making it a significant contender in the serverless computing arena.

Project architecture

Yoop is composed of two frontend applications, one for the landing, to improve SEO and for additional exposure, and one for the dashboard, considering it’s an authorization-based application which does not bring any SEO benefits, but might become significantly harder to maintain with SSR.

The backend is composed of various services, each with their individual purpose. The user has the possibility of creating an account. Upon authentication, the user will be allowed to create and manipulate their own functions. Authentication requests will go through the authentication service, along with the registration requests. Any other type of interaction, such as creating and editing functions or analytic real-time data, will go through a set of proxies, an HTTP proxy and a WebSocket proxy. The proxies will validate requests by sending data to the authentication service, specifically to the authorization endpoint of the service, and upon validation, the request will be forwarded to the correct services.

Whenever the user takes any type of action related to the function management (creation, updates, deletions), these requests will be forwarded to the function management service. The function manager will offload a configuration event to a RabbitMQ instance. The bootstrapping service will continuously check the queue for any events. Whenever new events are offloaded, the service will consume them and bootstrap an instance of a function, based on the config service. When successful, a new event will be produced by the bootstrapping service, signaling that the bootstrapping of the function was successful or it failed, and the status will be consumed by the function manager and will be sent back to the proxy, which will forward it back to the client.

The user will receive back an URL with a custom subdomain, bound to the primary domain of the platform and will be able to send HTTP requests to the URL received. The function can be written to handle the HTTP requests in any way the user wants.

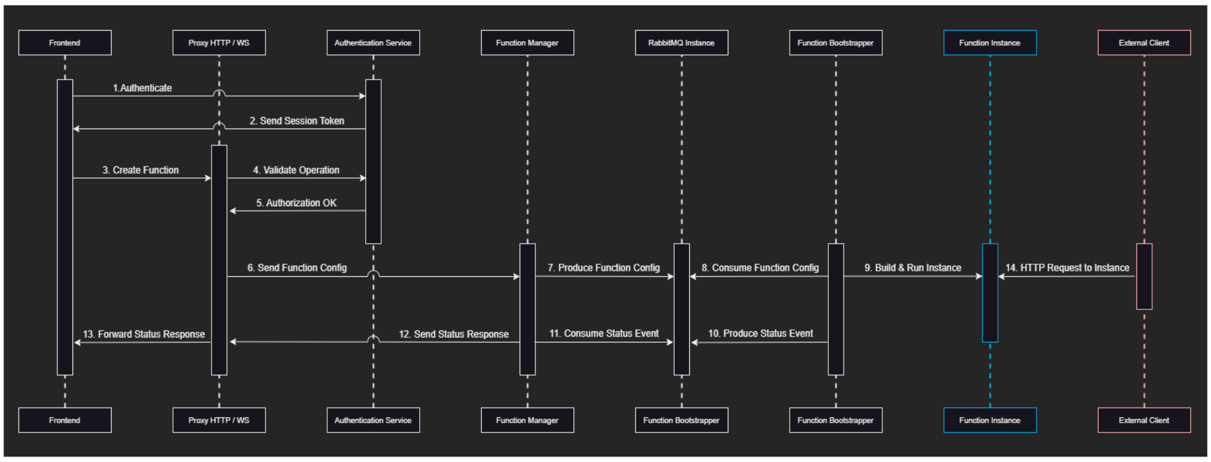

Figure 1 Application authentication workflow for creating a new function.
Author: Gabriel Muresan

The diagram above showcases a successful authentication flow, followed by a successful creation of a function, and the HTTP request coming from the consumer client, going to the new function instance of the user.

The application has only two actors, one which is a user, being able to authenticate or manipulate their own functions, the other being a function consumer, which can be another user building another function to call the original function, or somebody building a separate server, wanting to offload some processing to a separate process, or simply a client making a request to the function.

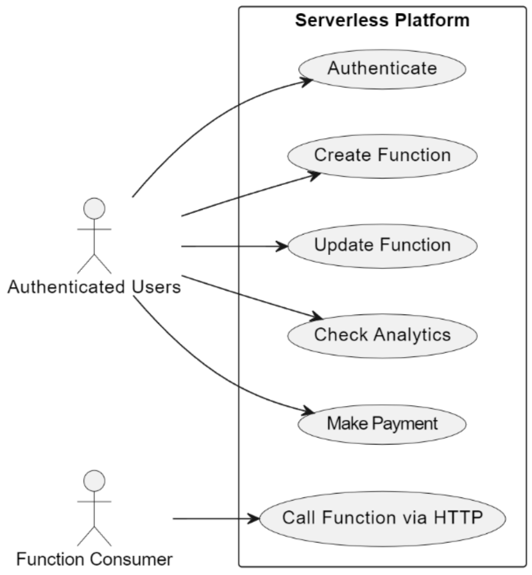

Figure 2 Use-Case Diagram
Author: Gabriel Muresan

The application architecture is straightforward and simple, making use of separate services built for specific purposes (e.g. proxies to handle specific requests and separate the client application from critical backend systems). There's a set of existing services which are deployed with database instances, like the function manager. The reason for deploying separate databases is to separate the load that a process can handle for different operations and for security purposes.

Implementation

The frontend of Yoop is engineered to provide a seamless and intuitive user interface, facilitating straightforward interactions with serverless functions. Utilizing Vue.js for its frontend development, the platform benefits from efficient DOM updates and smooth integration with diverse APIs and libraries. This architectural choice ensures that Yoop remains responsive and adaptable, minimizing disruptions during maintenance and updates while providing a dynamic environment for users to manage their serverless applications effectively.

On the backend, Yoop integrates advanced technologies such as NestJS, gRPC, and Kubernetes, establishing a robust infrastructure that supports secure, scalable, and efficient operations. This backend setup enables detailed management of user functions from creation through deployment, leveraging cloud-native technologies to ensure reliability and performance at scale. The comprehensive implementation details provided in this chapter highlight Yoop's commitment to building a powerful yet user-friendly serverless platform.

Frontend

The frontend of the Yoop platform is designed to offer a responsive, intuitive user experience while interacting with the serverless functions it supports. We developed the frontend using Vue.js, chosen for its efficiency in updating the DOM and its ease of integration with various APIs and libraries. Our frontend architecture separates concerns clearly between the user-facing interface and backend services, ensuring that updates and maintenance can be performed with minimal disruption to user activities.

Landing page

To optimize performance and user experience, we implemented advanced JavaScript techniques and utilized modern CSS frameworks for styling. The application is split into two main parts: the Landing page and the Dashboard, each serving distinct roles within the platform.

The Landing page of Yoop is crafted to provide the first point of contact for potential users. Its design focuses on clarity and speed, aiming to convey the essential information about our platform succinctly. The page highlights the advantages of Yoop, such as cost efficiency and ease of use, and provides clear call-to-action buttons to encourage user registration and engagement.

From a technical standpoint, the Landing page is built with Vue.js and uses lightweight components to ensure fast loading times even on slower internet connections. SEO is a priority for the Landing page, as we aim to rank well in search results to attract new users. We’ve implemented best practices for search engine optimization, including proper use of headings, metadata, and sitemaps.

The landing page outlines the ease and simplicity with which users can deploy serverless functions. It explains how Yoop automates the deployment process, allowing users to focus on writing code without worrying about the underlying infrastructure.

Potential users are introduced to the platform's robust analytics tools that help them track usage, monitor performance, and optimize costs effectively. This feature is vital for businesses to understand their application performance and make data-driven decisions.

Dashboard

The dashboard is a central interface that allows users to manage their serverless functions and gain insights into their performance. It is powered by Vue.js and is designed to offer an intuitive user experience that focuses on simplicity and functionality. The design prioritizes clarity and ease of use, allowing users to navigate through different features effortlessly.

The dashboard is built with Vue.js, which leverages its reactive data binding and component-based architecture to deliver dynamic and interactive user interfaces. Users can perform various functions such as creating, updating, and deleting functions. They can also access analytics insights and utilize the integrated development environment for code editing.

Integrated analytics tools provide users with valuable insights into function usage, performance metrics, and cost optimization. Visualizations such as charts and graphs enable users to track function invocation counts, response times, and resource consumption patterns. Predictive analytics features empower users to make data-driven decisions by forecasting function utilization and estimating associated costs.

Backend

The backend of our platform encompasses Authentication, Expenses, Payments, and Prediction Services, alongside the Function Bootstrapper and Kubernetes infrastructure. These components work in harmony to facilitate secure user access, efficient data management, and seamless deployment of serverless functions onto Kubernetes clusters. Together, they form the foundation of our platform, ensuring robust functionality and scalability.

Authentication, Expenses, Payments, and Prediction Services

We've adopted the "Code-First" approach in developing our services, prioritizing application logic and data access code over database-specific concerns. This methodology, aligned with agile development practices, promotes version-controlled schema changes using Git alongside application code. For our services, we've established four PostgreSQL databases: users, expenses, payments, and predictions, each dedicated to specific data categories to streamline access and ensure data integrity.

Our services are exposed through REST APIs, providing various endpoints for user interactions. Accessing these endpoints requires correct HTTP requests accompanied by a bearer JWT for authorization verification. Unauthorized access is restricted to login, token validation, and account creation functionalities. Account creation entails a POST request with mandatory fields including first name, last name, email, and password, with an optional phone number.

Upon receiving a valid request, the service verifies if the provided email is already associated with an existing account. If not, the service proceeds to create a new user account, capturing personal details such as first name, last name, and optionally, a phone number in the User database. Subsequently, the service creates user login credentials, storing the user's account ID, email, and password. Authentication is then performed, and upon successful completion, a JWT is generated and returned along with the user's account ID.

Other requests follow similar procedures, requiring a valid JWT for authorization. A validation protocol ensures the JWT's integrity, permitting requests only when the token is valid.

Function Bootstrapper

The function bootstrapper acts as a bridge between incoming HTTP requests from the Functions Manager and the deployment of serverless functions onto Kubernetes clusters. The following steps offer an overview of its implementation:

1. Initialization: The bootstrapper initializes a Kubernetes client to interact with the cluster. It sets up a connection to the Kubernetes API server using the provided Kubernetes config file or the default configuration.
2. Deployment Creation: When an HTTP request is received from the Functions Manager, the bootstrapper creates a Kubernetes deployment for the function specified in the request. It sets up the deployment with the necessary configurations, such as the container image, ports, and replicas. Additionally, it creates a corresponding service to expose the function internally.
3. Function Code Update: As part of the deployment process, the bootstrapper updates the function code inside the deployment's containers. It fetches the code from the HTTP request and injects it into the appropriate location within the container filesystem. This ensures that the deployed function contains the latest code changes.
4. Deployment Management: The bootstrapper provides functionalities for managing deployments, including listing existing deployments, retrieving details of a specific deployment, and deleting

deployments when necessary. These operations enable efficient lifecycle management of serverless functions within the Kubernetes cluster.

The function bootstrapper serves as the orchestrator for deploying serverless functions onto Kubernetes clusters. It handles HTTP request ingestion from the Functions Manager, creates and updates deployments, and manages their lifecycle to ensure the seamless operation of the serverless function platform.

Kubernetes and related services

It is important to understand the architecture and components of a Kubernetes deployment. A cluster encompasses everything. It consists of a set of worker machines, which are called nodes. Obviously, in a cluster there is at least one (worker) node. Next, each node hosts pods. The pods support the application workload. In our case, using Docker, each pod can contain multiple Docker containers, but the contents of a pod don't modify after deployment. A container is the smallest unit we are interested in from an infrastructure point of view. A service is used for exposing an application running on one or mode pods.

In our case, a pod is allocated for each function. Each pod only needs to contain one Docker container, which runs the function. Whenever the resource usage on one pod reaches a certain threshold (for example 80% CPU utilization), a new pod, containing the exact same container is created. This is called a replica. All replicas hosting the same function are behind one service. There is one service per function.

A practical example is local development/ testing. When Kubernetes starts, there is always one cluster. Locally, there is one node (the workstation of the developer). Let's say three functions are defined: A, B and C. Thus, three services exist on the node. These can be service A, B and C for the respective functions. Initially three pods exist on the node, one behind each service. Each pod hosts one container running their respective functions. Then, function A is heavily requested, function B has some serious usage, but not as much as function A and then function C has almost no traffic. In this scenario, behind service A might be 5 replica pods hosting function A, service B will have 2 pods and finally service C only has one pod as there is no need to spawn more. The load balancing takes place on each service, among the different identical pods (replicas) behind it.

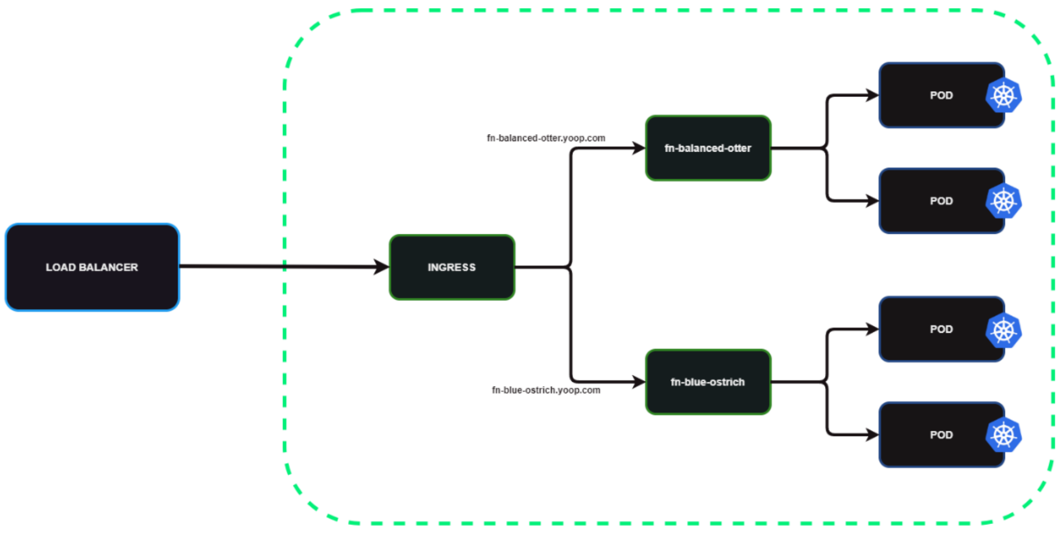

Figure 3 Name-based virtual hosting.
Author: Rares Chiuzbaian

Nginx Ingress facilitates access from outside the cluster to all these resources. Each function has a subdomain that the client’s access, from which the traffic is routed to the correct pod based on the Ingress rules. Then, the load balancer has the task of routing the request to one of the existing containers or signaling the need to spawn one more.

All these interactions are logged, and the data is then streamlined by Prometheus. Elements monitored include the number of requests for each subdomain (thus each function), the resource usage of different pods and so on.

This data can easily be viewed in interactive dashboards facilitated by Grafana and used by our machine learning service for future usage predictions and analytics.

Conclusions

The development of the Yoop platform represents a significant advancement in the serverless computing space, providing a versatile and user-friendly environment that caters to both novice and experienced developers. This platform differentiates itself from existing solutions like ServiceNow Flows and AWS Lambda by offering a dual approach that combines the simplicity of block-based programming with the robustness of custom code development.

Yoop stands out for its focus on affordability and accessibility. Unlike mainstream cloud solutions that may present cost barriers to smaller entities, Yoop is designed to be cost-effective, making it an attractive option for startups and individual developers who require a reliable serverless environment without significant financial outlay. This approach not only democratizes access to powerful computing resources but also encourages innovation and experimentation.

The integration of technologies like Vue.js, NestJS, and Kubernetes within Yoop's architecture ensures that the platform is both scalable and secure. These technologies provide a solid foundation for handling high volumes of traffic and complex computations while maintaining system integrity and responsiveness.

The platform's emphasis on analytics and predictive features offers users critical insights into function performance and cost management. This proactive approach allows users to optimize their applications efficiently, ensuring that resources are used judiciously and that applications remain cost-effective over time.

Yoop provides a comprehensive serverless solution that bridges the gap between ease of use and advanced functionality. By enabling users to seamlessly transition from block-based setups to more sophisticated coding environments, Yoop not only enhances the serverless computing landscape but also fosters a more inclusive and innovative development community. As the platform evolves, it is poised to introduce more enhancements and integrations that will further solidify its position as a leading tool in serverless computing.

Bibliography

[1] "Why Astro?," [Online]. Available: https://docs.astro.build/en/concepts/why-astro/. [Accessed 10 May 2024].

[2] "Introduction | Vue.js," [Online]. Available: https://vuejs.org/guide/introduction.html. [Accessed 4 March 2024].

[3] "API Reference - Streamlit Docs," [Online]. Available: https://docs.streamlit.io/develop/api-reference. [Accessed 6 May 2024].

[4] "Deno Runtime Quick Start | Deno Docs," [Online]. Available: https://docs.deno.com/runtime/manual. [Accessed 23 March 2024].

[5] "Documentation - The Go Programming Language," [Online]. Available: https://go.dev/doc/. [Accessed 14 May 2024].

[6] "What is .NET? An open-source developer platform.," [Online]. Available: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet. [Accessed 14 March 2024].

[7] "Kubernetes Components," [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 13 May 2024].

[8] "Ingress," [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/. [Accessed 5 March 2024].

[9] "Overview | Prometheus," [Online]. Available: https://prometheus.io/docs/introduction/overview/. [Accessed 13 May 2024].