symmetry

MDPI

Article

Multipurpose Cloud-Based Compiler Based on Microservice Architecture and Container Orchestration

Sayed Moeid Heidari *,† and Alexey A. Paznikov *,†

Department of Computer Science and Engineering, ul. Professora Popova 5, 197022 St. Petersburg, Russia
* Correspondence: skheydari@stud.etu.ru (S.M.H.); apaznikov@gmail.com (A.A.P.)
† These authors contributed equally to this work.

Abstract: Compilation often takes a long time, especially for large projects or when identifying better optimization options. Currently, compilers are mainly installed on local machines and used as standalone software. Despite the availability of several online compilers, they do not offer an efficient all-in-one package for private account management, command line interface (CLI), code advisors, and optimization techniques. Today, the widespread usage of Software as a Service (SaaS) is ever-growing, and compilers are not an exception. In this paper, we describe a symmetric approach to compilation and how to compile code on distributed systems. Although some improvements in cloud compilers have been made, it is possible to harness the potential of the most-modern technologies and architecture patterns toward designing efficient, in-cloud compilers. In this paper, we propose an architecture design of a cloud-based compiler that is fully compatible with orchestration technologies, such as Kubernetes, providing a higher level of scalability, reliability, security, and maintainability. Microservice architecture alongside containerization and orchestration technologies assist us in making a scalable system that provides a high level of availability. We propose this architecture so that the system can handle a higher workload as it receives a large number of compilation requests per second. Distributed compilation is a prominent benefit of this approach, as each phase of the compilation can be executed in a separate server, which supplies a kind of workload mitigation to the whole system. In other words, we propose a new perspective for an intelligent way of advisor, error detection, and optimization of compilers. We also propose an implementation example of the developed architecture. Finally, we analyze the results from an experimental implementation, proving that we can compile code from more than 100k requests concurrently on a cloud cluster with one master node and three worker nodes.

Keywords: compilation; optimization; distributed systems; cloud computing; Kubernetes; microservice architecture

Citation: Heidari, S.M.; Paznikov, A.A. Multipurpose Cloud-Based Compiler Based on Microservice Architecture and Container Orchestration. *Symmetry* **2022**, *14*, 1818. https://doi.org/10.3390/sym14091818

Academic Editor: Jan Awrejcewicz

Received: 10 August 2022
Accepted: 28 August 2022
Published: 2 September 2022

1. Introduction

The amount of data to be processed, the complexity of algorithms, data collection to feed neural network training [1], hardware resource restrictions, and new optimization techniques for new microarchitectures lead us toward a new approach to data processing and highly accessible computing resources. Thus, the idea of migrating compilers to the cloud environment and deploying them as a microservice architecture is quite promising [2].

With the current situation (high complexity of centralized, monolithic, and standalone applications) of compilers, the whole source code goes to the compilation pipeline as soon as the compiler starts to compile.

The list below reveals some other restrictions with the current situation of compilers as monolithic applications that the idea proposed in this paper is going to solve.

- Limited hardware resources;
- Parallel compilation (we can perform some compilation stages in parallel, but we do not take advantage of this opportunity);

- New phases of compilation should happen inside the compiler;
- Microarchitecture-specific code generation and optimization;
- Compiler versioning;
- Testing limitations in compiler development: TDD (test-driven development) and BDD (behavior-driven development)

We focus on these issues in Section 1.1.

The usage of cloud computing and cloud dependencies is ever-growing nowadays. According to the Haptic [3] analysis, European countries have almost 26% of the share of cloud computing use. With cloud computing, it is possible to automate software updates and integration specifically in microservice architectures. Cloud computing also brings efficiency, cost reduction, automated horizontal and vertical scalability, disaster recovery, mobility, and data-loss prevention.

Bringing compilers to the cloud environment will provide a multi-purpose and reliable system for different types of compilers. It will also make compiler development much faster and easier to control.

To summarize, we make the following contributions:

- Provided a method to port compilers to a cloud environment;
- Containerize different modules and the compiler;
- Provide a network layer in front of the modules inside the container;
- Provide a CLI tool to communicate with the compiler on the cloud;
- Apply service discovery on the cloud cluster to facilitate services communication.

In Section 1.1, we describe the current state of compilers and their limitations. In Section 2 we analyze related works. We introduce the proposed microservice approach in Section 4. Then, the technical description of this approach is described in Section 5. The implementation details of compiler deployment in a cloud environment are described in Section 7. We present the results of our experiments in Section 8. We discuss the provided approach in Section 6. Finally, in Section 9, we provide a conclusion and a discussion about future work.

1.1. Limitations of the Current State

Hardware plays a crucial role in compilation and code execution, as more-powerful hardware resources make the compilation process faster and more accurate. For example, cache memory usage, pipelining, out-of-order execution, simultaneous multithreading, vectorization, accelerators, and others technologies can be optimized by a compiler only with the full awareness of the machine architecture.

As an example, 12th generation Intel Core i9 (Alder Lake) processors have the following most-highlighted technologies: Gaussian and Neural Accelerator, Thread Director, Deep Learning Boost (Intel DL Boost), Optane Memory Support, Speed Shift Technology, Turbo Boost Max Technology 3.0, Turbo Boost Technology, Hyper-Threading Technology, Virtualization Technology (VT-x), Virtualization Technology for Directed I/O (VT-d), and VT-x with Extended Page Tables (EPT).

Having the opportunity to use these technologies may not be possible for many users or developers because they might not have the most-recent hardware. Thus, they cannot compile their source code according to the hardware on which it will be executed. With the proposed idea, the most-recent technologies can be provided to users in a shared environment.

1.2. Parallelization of Compilation

The compilation process is naturally sequential (each phase happens just after the previous one), as the input of each stage fully depends on the output of the previous one. Further, it is almost impossible to use parts of a compiler concurrently with other tasks when it is busy with other compilations by multiple users or applications. This means when a compiler is processing code, practically speaking, we cannot use modules for other

purposes [4]. With architecture provided on the cloud, we can use each module of the compiler at any time. As an example, when a lexical analyzer is parsing code and busy with Abstract Syntax Tree (AST) generation, we can use code generation as an independent microservice with another intermediate representation (IR) as input. Figure 1 below shows such a scenario.

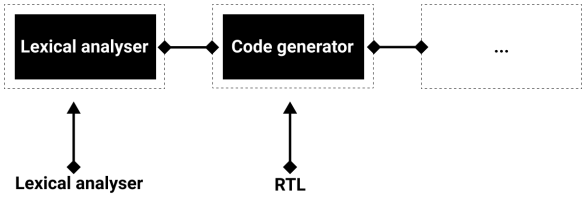

Figure 1. Compiler multi-usage.

1.3. Adding New Compilation Phases

If we add new plugins, passes, and modules for a compilation, we have to use the same programming language that the compiler source code has been written in. For example, if we want to create a new plugin for the GCC compiler, we have to use C or C++ to create a shared library [5]. Thus, this is a restriction. In our method, however, any programming language or technology can be used because it is going to act as a completely independent microservice. In this approach, we can take the output of a pass and pass it as an input to another microservice to do some processes, and then take its output and send it to another microservice that is written in a completely different language and technology [6]. Later, in Section 1.5, we talk about this technique in detail.

It is possible to add new phases to compilers by making new plugins or passes as separate modules that can be loaded by a compiler. These plugins and passes need to be used inside the main compilation process and, indeed, they are injected into the main compilation pipeline. Figure 2 below shows a hierarchy of GCC plugin modules and libraries that can be used to create custom plugins [7].

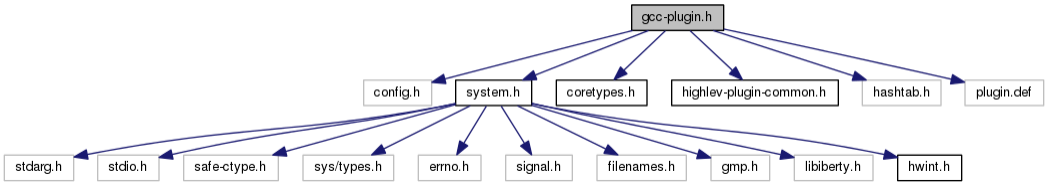

Figure 2. GCC plugin class hierarchy.

To produce such new passes, we need to have access to the source code of a compiler, then rebuild it with the pass manager. This may be a time-consuming process causing various issues (such as different versions of a compiler). This can be solved by a SaaS (Software as a Service) approach, By introducing new pods (the smallest execution unit in Kubernetes's model) to the cluster worker nodes from a separate microservice. New passes can be written in any programming language and independently, because microservices are technology independent. Then, we can use the new microservice for any stage of the compilation process. Thus, it is not limited to the functionalities of the compiler core.

1.4. Code Generation and Optimization

When source code is compiled on a local computer, the machine code is generated by the compiler specifically for that machine's architecture and operating system (OS) [8]. With our approach, this problem can be solved as a user can choose the most desirable architecture or OS for which the code needs to be compiled. The cloud service can provide a vast range of architectures and OS types based on account settings [9].

1.5. Technology Independence

Compilers are mostly written in low-level languages, and any contribution to them should happen at the same level. The back-end of the compiler is responsible for generating the actual machine code. Thus, any pass in the back-end should be written in a low-level language due to the higher accessibility to the hardware and processor level. There are still some parts of the compiler, specifically the front-end and middle-end, that can function in a higher language or technology. For instance, suppose we need to add a new phase to the compiler that can analyze an IR code, prepare a report, and generate advice, or we need to apply some transformations.

With the current structure of compilers, it is not possible to involve a different technology in the compilation process. One of the problems that hs been solved by our proposed architecture is the possibility of using any low or high level technology in the compilation time. By using such a microservice-based architecture, the modules of a compiler can be written in different technologies. Consequently, the whole compiling system can be expanded regardless of the technology or language.

LLVM, as a collection of modular and reusable compiler and tool-chain technologies, helps us to create new front-ends for the specific language. The front-end reads the actual source code and after lexical and syntax analysis, it generates the first IR for the middle-end. Afterward, optimizer in the middle-end apply possible optimization techniques on the IR-level and then generate the first and second transformed IRs. At the last stage, this transformed IR code is passed to the back-end to generate the actual machine code. Figure 3 shows such a process.

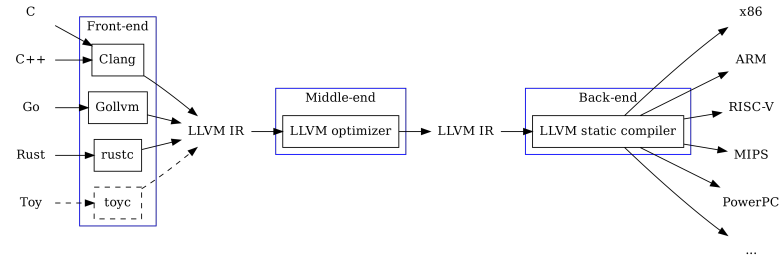

Figure 3. LLVM compiler pipeline.

LLVM is quite flexible due to the separation of the front-end, middle-end, and back-end. Due to this flexibility, it is quite possible to write a fully customized sanitizer for a specific language. We can write a sanitizer as a pass in LLVM to modify or analyze IR tuples. LLVM pass manager is responsible for attaching the newly written sanitizer as a pass to the compilation process. Figure 4 shows such a process.

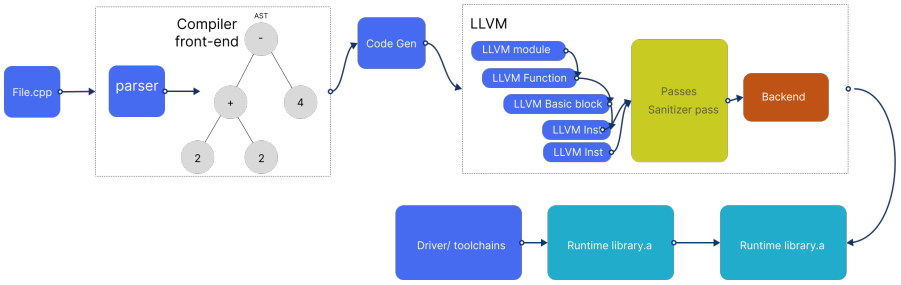

Figure 4. LLVM sanitizer pass.

As we can see, the sanitizer pass is tightly coupled with the LLVM tool-chain. With the proposed architecture, we break down this dependency and make it possible to write sanitizer passes by a fully separated microservice. Even though we have the pass in the middle of the process, we simply use it to send the instruction to the out of the LLVM tool-chain process. Then we return it to the compilation pipeline, as shown in Figure 5.

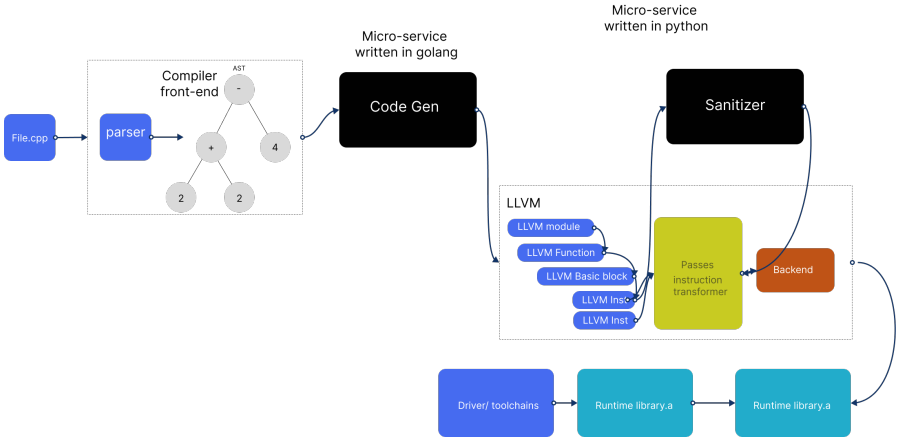

Figure 5. Sanitizer with microservice architecture.

According to this idea, it is possible to involve new microservices in the compiling process and to use other languages rather than just low-level languages such as C/C++. As another example, we can write new front-ends to parse a new language and generate the AST based on grammar that is written in new technology, and introduce it to the LLVM pass manager.

1.6. Testing Limitations in Compiler Development

Automated tests, one of the vital steps in CI/CD (Continuous Integration/Continuous Deployment) of the software development process, may sometimes be quite difficult and time-consuming with the current compilation process. To implement automated tests in compilers, we need to always run them over the whole application. By porting compilers from a monolithic architecture to microservice architecture, tests of a single microservice need to be executed only over that single microservice and not over the whole application. Because these two processes are completely independent.

We have prepared a fully automated CI/CD pipeline for this purpose. Each microservice is hosted on a separate repository. The repositories are connected to our builder system (Jenkins) as multi-branch pipelines. With these multi-branch pipelines, we have different pipelines for different branches. Overall, there are two major branches: Develop and Main. Bypassing the Develop branch pipeline, we tag the containers as "develop" and push them to a private Docker registry. From the other side, we deploy and version the containers from the Main branch with the tag "stable" and deploy them in the production environment. Figure 6 reveals such a process.

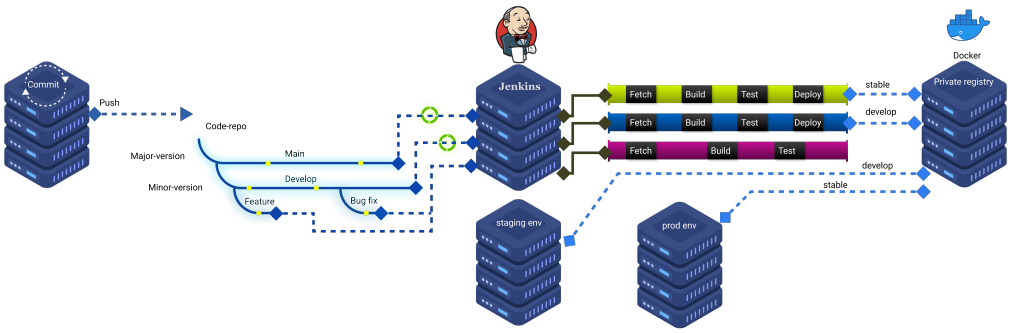

Figure 6. Deployment pipeline.

We produced containers containing LLVM modules with an HTTP front layer and then prepared an automated CI/CD pipeline on Jenkins, so that by pushing new codes to the repository, we hook the builder on Jenkins and start a pipeline for test and deployment automation. Figure 7 is an example of such a pipeline.

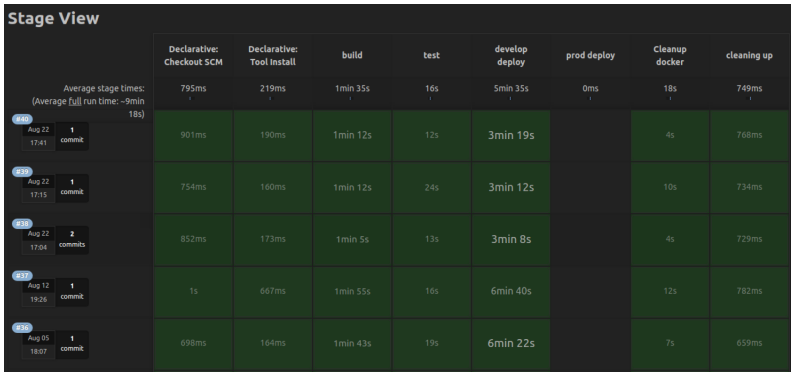

Figure 7. CI/CD pipeline on Jenkins.

This pipeline has eight stages overall. The first tree stages fetch the code from the repository and build the project. The fourth stage tries to run the tests over the built project inside the container. Such separation of services and branches makes a safe environment for testing, quality assurance, and reliable deployment. If any bug happens with one of the services, we still have the latest stable version in the production environment, and we simply catch the bug in the staging environment.

Afterward, the next stages try to push the tested container to the docker registry to make it accessible for the production environments. After this process, all of the modules are deployed on the Kubernetes cluster as an orchestration of the microservices. Figure 8 is a schematic of such a resource map on Kubernetes.

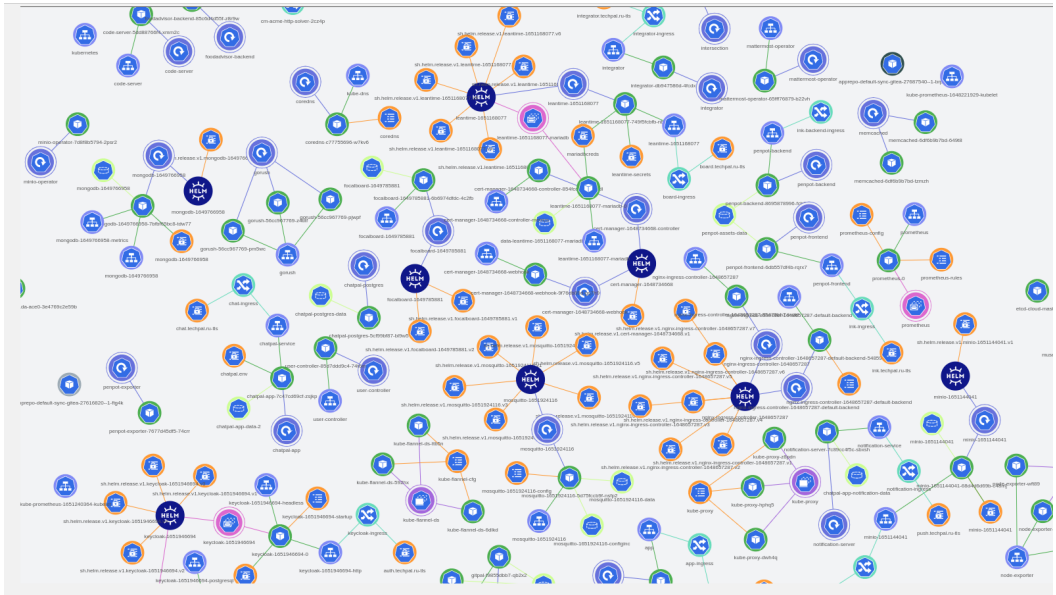

Figure 8. Deployment resource map.

2. Related Works

The authors of [10] proposed deploying a compiler on a private cloud that can be established and managed inside a company's internal network. In this paper, the authors proposed an architecture that takes the code from a client and, after compiling it, produces an executable file and generates a direct link to it. Most users do not prefer to store their execution files on a third-party server, and storage issues also appear. Storing executable files or compilation history should be an option for the user, and the authors of this paper did not point to such an option, which can also raise security and privacy issues.

The authors of [11] proposed an architecture of the compiler service and the deployment model. Their approach consisted of the same monolithic compiler installed on the

server, and there is no module destruction. In this paper, the authors provided an algorithm to dedicate servers to incoming requests by a priority list. In this approach, if the servers are all busy, then the user waits for a server to be free. In other words, there is no vertical-scaling mechanism, and the servers are statically allocated for a compiler service. The approach provided in this paper is completely monolithic, and there is no mechanism for compiler module separation.

The authors of [12] introduced a theory in which a shared pool of configurable computing resources is provided. The model is proposed based on a private cloud model provisioned on Ubuntu Enterprise Cloud (UEC). It provides hosted services to a limited number of clients, and the service is distributed in a heterogeneous manner [13]. In such a model, the authors provided the resources of a cloud computing system (CCS) with multi-state devices.

As a sufficient evaluation of such a system, the authors proposed a non-sequential Monte Carlo simulation (MCS) with a traceable approach to assessment by iteratively drawing many random samples and observing system behavior. After this assessment, there is also a classification step of MCS, in which the requested and available system resources with the device dimensions are compared. At the same time, the utilization of each device is calculated.

M. Pabitha, T. Selvakumar, and S. P. Devi also proposed a compiler over a private cloud on a Linux environment in [14]. In this paper, we can see how a cloud environment can be prepared to serve a compiler as a service. Although it runs the compiler over a cloud infrastructure, it still uses the same standalone monolithic compiler, and there are no scalability or high-availability features. It is also accessible for only a limited number of users. In addition, it does not provide a replication management system, and the whole system dies if the compiler service stops responding for any reason.

S. Taherizadeh and M. Grobelnik [15] provided a set of key factors to consider in the development of auto-scaling methods. In their paper, the three key influencing factors including conservative constants, adaptation interval called control loop period (CLTP), and stopping at a maximum of one container instance in each CLTP have been introduced.

The authors provided a method to tune the auto-scaling of containerized applications under the condition of predictable bursting workloads. The CPU-based auto-scaling policy of Kubernetes causes some container miss-instantiating problems with minor fluctuations of the containers. The authors of this paper also provided one further step to be considered: various adaptation intervals rather than a fixed period by the Kubernetes CLTP. Most cloud auto-scaling practices are done according to infrastructure-level rules, such as CPU-based auto-scaling policies which are related to the average of the CPU utilization threshold. For example, if the threshold is set at 80%, a new replication will be created at 81% utilization. The authors of this paper proposed an algorithm with a conservation constant (a) that is the constant for auto-scaling.

The authors of [16] examined some compilers and evaluated the generated code from each of them with respect to the ISA (instruction-set architecture). They examined some parameters of the code generated by the compilers, including dynamic instruction count, performance, generated code size, power consumption, and execution time. Using different compilers can result in large performance values even on the same target machine. Thus, selecting a compiler for system development is very important to get the highest performance. The authors did a detailed comparison of LLVM and GCC with respect to code size and dynamic instruction count for EISC-embedded processors. Afterward, they compared different architectures, such as RISC, CISC, and EISC based on the EEMBC benchmark (including representatives of various kinds of embedded applications). The experiments were done with the "-O2" optimization option. Their results showed a best-case of LLVM reducing dynamic instructions by about 80% in iirflt benchmark. For autocor, the dynamic instruction count increased by 70%, and LLVM outperformed GCC in some benchmarks such as iirflt and bezier-fixed. LLVM performs better in loop-unrolling. An innermost loop is completely unrolled even if it has a loop unrolling factor of eight. Consequently, GCC

and LLVM are the most popular compilers, and the codes generated with them have the highest performance.

3. Cloud Infrastructure

Cloud computing architecture consists of a number of coupled distinct components. Generally speaking, cloud architecture can be divided into two main parts: front-end and back-end, which are connected through a network [17].

The front-end of the cloud architecture refers to the user side of the cloud computing system. It is made up of an interface and applications that are useful for accessing and managing the cloud platform (usually a web application) [18].

The back-end of a cloud system refers to the cloud system on the server. It has all the resources required to provide cloud computing services. It comprises a huge amount of data storage, virtual machines, services, security firewalls, deployments, pods, stateful sets, and replica sets. It is worth noticing that the built-in security mechanism, traffic control, and protocols are the responsibilities of the back-end of the cloud computing system. The cloud server employs a set of protocols referred to as middleware that helps the services communicate with each other.

3.1. Cloud Computing

Cloud computing is a kind of computation that involves several computers located in different locations around the world connected to the main computer (accessible within the network) through any accessible network [19]. Cloud computing provides on-demand network access to a shared pool of configurable computing machines (e.g., computing resources, networks, storage environments, and applications). Services on the cloud are classified as Software as a Service (SaaS), Communication as a Service (CaaS), Platform as a Service (PaaS), Infrastructure as a service (Iaas), or Network as a service (NaaS).

Deploying the services on a cloud infrastructure reduces the overhead and cost for the end-user while increasing flexibility.

3.2. Virtualization Concept

Sharing a single physical instance of a machine or application among several users is referred to as virtualization [20]. It happens by assigning a logical name to a dedicated physical resource and providing a pointer to that physical resource on demand. A virtual machine (VM) provides a virtual environment that is separated from the underlying hardware logically. The machine hosting the virtual machine is called the "host machine", and the guest machine is referred to as the "virtual environment" on the host machine. The technology that makes it possible to provision a virtual machine on a host machine is called a hypervisor. Since hypervisors are a kind of software layer by which a host computer can support multiple VMs simultaneously, they became a key element of the technology that makes cloud computing possible.

In general, there are two different types of hypervisors: bare metal, which executes on a bare system and handles virtualization tasks directly onto the hardware before the system; and hosted, which runs within the operating system of the host machine and can emulate the devices of a VM.

3.3. Containerization and Microservices

Today, encapsulation of an application as a single and independent package of software that can bundle application codes together with all of the dependencies, libraries, and configuration files needed to run is possible with containerization. To do so, run-time engines (e.g., Docker) should be installed on each node of a cloud cluster (master and worker nodes), which creates a condition for the containers to share a single operating system kernel inside the same node (independent server) with all other containers. Figure 9 represents an example of a containerized application that contains several independent microservices and open-source services.

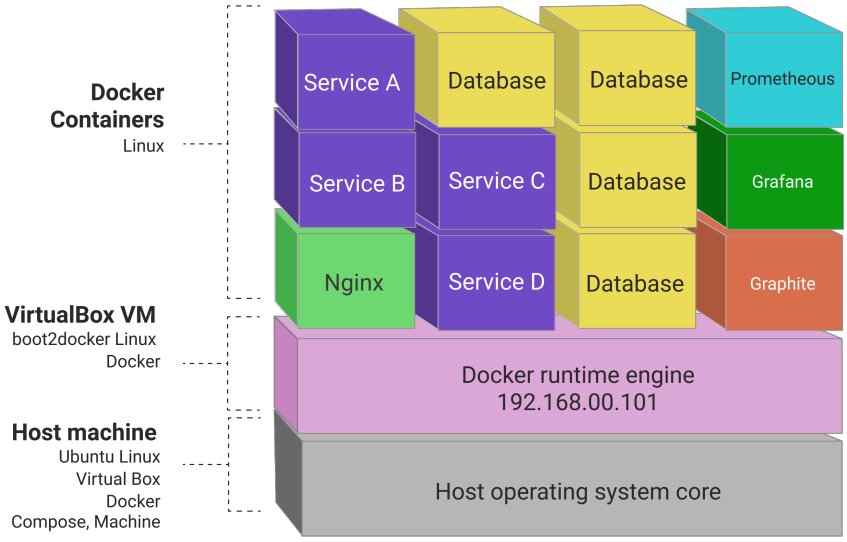

Figure 9. Containerized microservices.

A containerized application can contain hundreds or thousands of containers. This issue introduces overwhelming complexity if managed manually. Container orchestration handles this complexity and makes it more manageable, as it provides a declarative approach to automation.

Finally, microservices are an architectural approach to software development that structures software as a network of small and independent services that communicate with one another [21].

In classical monolithic architectures, the whole application runs as a single service so that all of its processes are tightly coupled [22]. Thus, if a process of an application has an issue or a spike in demand, the entire architecture should be redesigned and scaled. As the code base grows, improvement of a monolithic application's features become more convoluted, similar to current compilers installed on clients' machines locally or in the cloud. Based on microservice architecture, an application is built as small and independent components that can execute each application process as a service. Each service is responsible for business logic and performs a single functionality. Due to this independence, each service can be updated, deployed, and scaled to meet the demand for a specific feature [23].

The execution model in current compilers such as GCC is quite straightforward, and a single application is responsible for all features of the compiler (Figure 10).

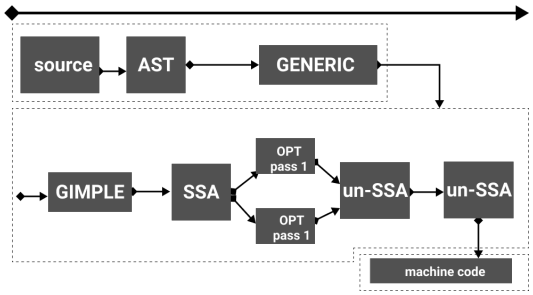

Figure 10. GCC as an example of a monolithic compiler model.

4. Proposed Approach

In the proposed approach, we represent the phases of compilation as different microservices plus include other microservices such as a linker, assembler, code advisor, database controller, and user manager (Figure 11).

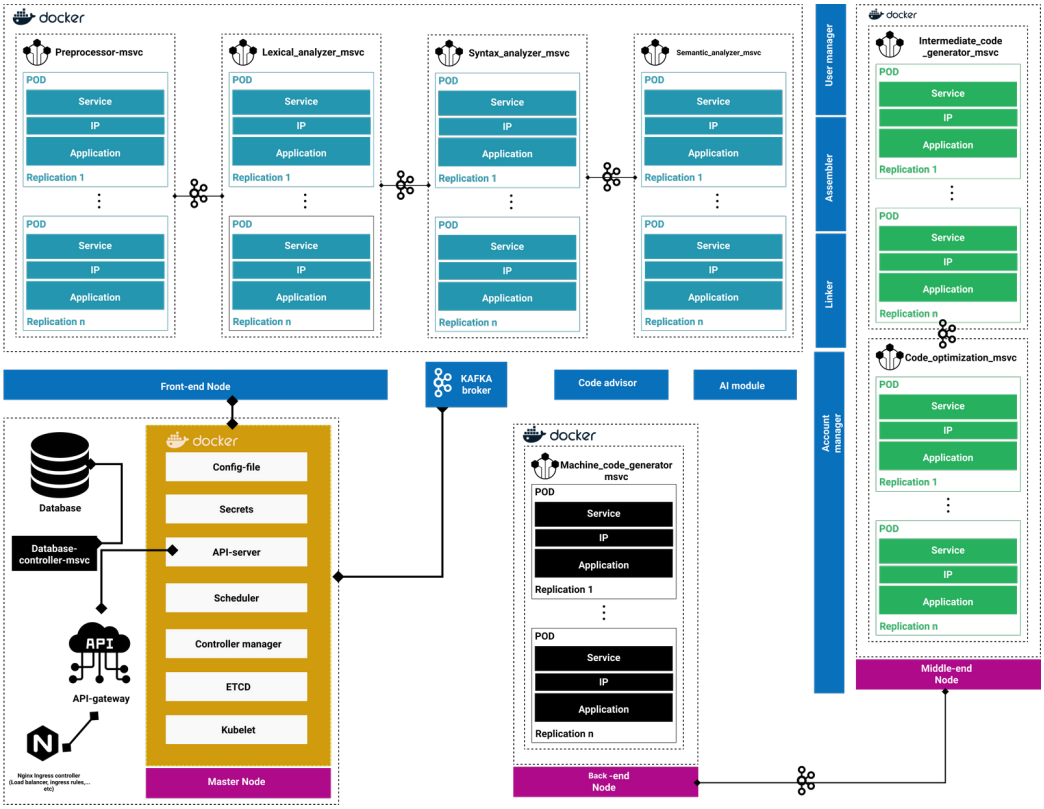

Figure 11. Microservice compiler model.

In this model, each phase of compilation is executed separately as an independent service. The output of each microservice is transferred to the Kafka microservice (distributed event streaming platform). The aforementioned architecture focuses on Google Kubernetes container orchestration technology [24]. Kubernetes is an open-source project that provides container orchestration to provision multi-node cloud clusters. With this technology, we deploy the microservices as independent containers using Docker as a containerization platform. With this model, we can have hundreds or even thousands of container replications (pods).

Each microservice can have many copies of one pod through a system called "replica set". If a pod dies, there are other replications of the same pod to handle the new incoming requests. And another service called "Kubelet" raises a new pod to replace the died one. These mechanism helps the system to improve the availability of the whole orchestra as well as mitigate downtime.

4.1. Inter-Service Communication

In the monolithic style of current compilers, the application is tightly coupled, and all the layers of the application are accessible to the user as a bottleneck. Thus, there is no need for inter-service communication (Figure 12).

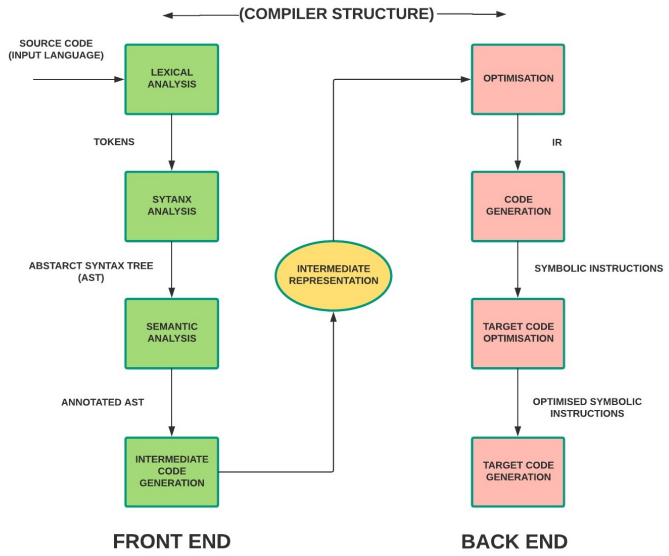

Figure 12. Current compiler architecture.

When it comes to microservice architecture, we need to divide the application into multiple services, and each service has its own storage environment. In our architecture, we use producer–consumer-based Kafka broker (Figure 13) as our microservice communication technology so that each service has full connection with other services. Instead of sending data between different modules of the compiler in shared memory, we send them through a network protocol (TCP/UDP or HTTP/HTTPS). For each communication scenario, we have a defined message pattern as illustrated in Figure 13.

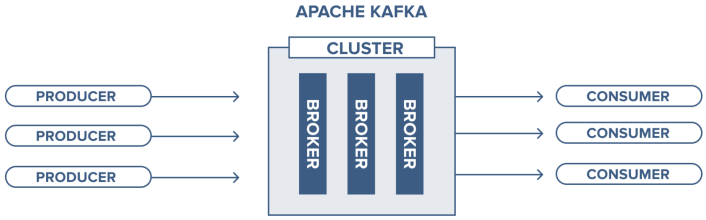

Figure 13. Kafka structure.

Further, we divide the front-end and back-end of the compiler into two different namespaces in the cloud environment. Each namespace has its own responsibility, and the results of the first namespace go to the next namespace through the Kafka broker. An internally accessible service with a cluster IP type (a service that is accessible internally) associated with a defined IP is dedicated to a group of pod replications. Each deployment can be served to a large number of users.

In our microservice architecture, communication between services plays a crucial role on performance. Thus, according to our requirements, we need to choose the right approach for inter-service communication.

4.2. Services Routing

We use Nginx as our reverse proxy in the implementation of Kubernetes. ingress [25] acts as a load balancer in the front line of the cluster. The first destination of the incoming requests will be our API gateway, which takes the requests, passes them through a security layer, and spreads them among the microservices.

Figure 14 shows the approach in detail.

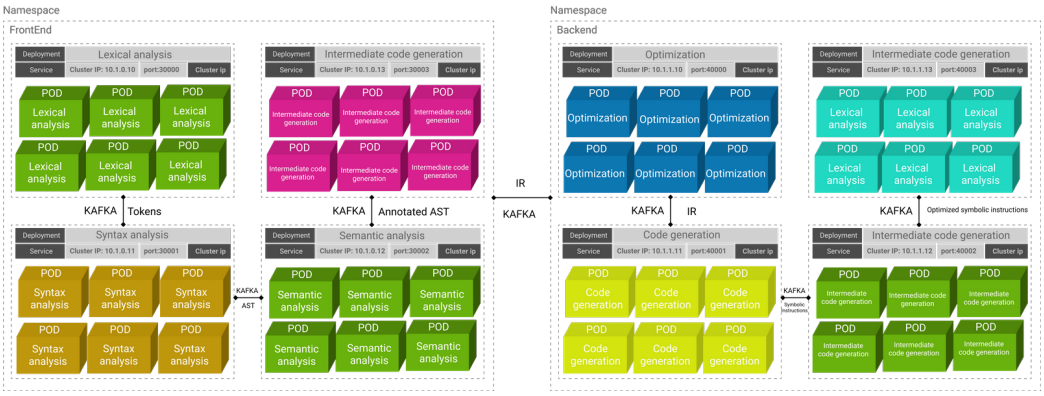

Figure 14. Cloud compiler inter-service communication model.

5. Technical Description

In this article, we use LLVM as a collection of modular and reusable compiler and tool-chain technologies for our implementation and experiments. To develop such a system, we need to have a front layer for each LLVM (compiler infrastructure as a collection of modular and reusable compiler and tool-chain technologies) tool that is going to provide services to the end-user in HTTP protocol. By providing APIs for each feature, the gateway layer of our microservice implementation makes a command request in the background to the appropriate LLVM tool and sends back the result as an HTTP response to the requesting service. Figure 15 below provides a scheme of this mechanism.

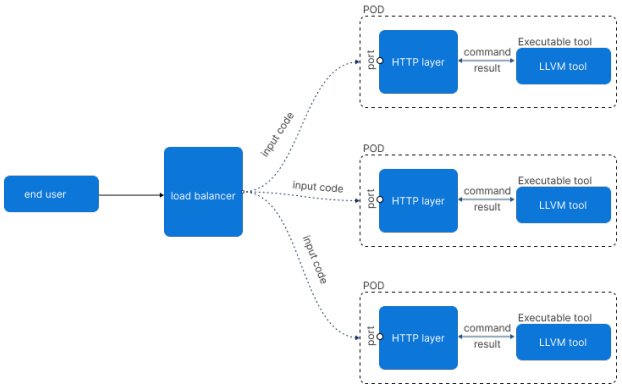

Figure 15. Compile LLVM tools.

LLVM provides several tools separately for each phase of the compilation. For this purpose, we can build each part of LLVM with its appropriate object files and shared libraries as shown below in Figure 16.

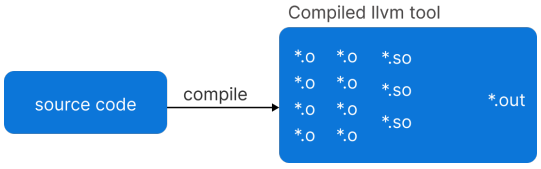

Figure 16. HTTP front layer.

After building a tool with LLVM (each tool will be a different compilation phase) we prepare an HTTP layer that acts as a front layer for incoming HTTP requests by listening on a specific port. As an example, an HTTP layer can take a request from a service that transforms an input file containing human-readable LLVM assembly language, translates it to LLVM bitcode, and writes the result into a file or to standard output. We can do such an operation using llvm-as (LLVM assembler tool) [26].

Below is a short list of available LLVM tools that can be packaged as Docker containers to be deployed as pods inside a Kubernetes cluster.

- dsymutil—manipulate archived DWARF debug symbol files
- llc—LLVM static compiler
- lli—directly execute programs from LLVM bitcode
- llvm-as—LLVM assembler
- llvm-config—Print LLVM compilation options
- llvm-cov—emit coverage information
- llvm-cxxmap—Mangled name remapping tool
- llvm-diff—LLVM structural 'diff'
- llvm-dis—LLVM disassembler
- clang-tidy syntax analyzer
- llvm-dwarfdump—dump and verify DWARF debug information
- llvm-lib—LLVM lib.exe compatible library tool
- llvm-libtool-darwin—LLVM tool for creating libraries for Darwin
- llvm-link—LLVM bitcode linker
- llvm-lipo—LLVM tool for manipulating universal binaries
- llvm-mca—LLVM Machine Code Analyzer
- llvm-otool—Mach-O dumping tool
- llvm-profdata—profile data tool
- llvm-readobj—LLVM Object Reader
- llvm-remark-size-diff—diff size remarks
- llvm-stress—generate random .ll files
- llvm-symbolizer—convert addresses into source code locations
- opt—LLVM optimizer
- llvm-addr2line—a drop-in replacement for addr2line
- llvm-ar—LLVM archiver
- llvm-cxxfilt—LLVM symbol name demangler
- llvm-bcanalyzer—LLVM bitcode analyzer
- FileCheck—Flexible pattern matching file verifier
- llvm-ifs—shared object stubbing tool
- llvm-profgen—LLVM SPGO profile generation tool
- llvm-tli-checker—TargetLibraryInfo vs. library checker

6. Discussion and Limitations

As compilers, and specifically LLVM/Clang, are not naturally microservice-based, we need to consider the complexity of the conversion. The proposed approach describes how it would be possible to migrate the compiler to a cloud environment. With the compiler as a microservice, we have the opportunity to scale up and down the replicas of the microservices (Vertical Scaling). However, Horizontal Scaling also can bring some benefits. With Horizontal Scaling, we can improve the power of the nodes or join new nodes to the system (as worker nodes); we can maintain the compiler as a service with high availability.

Deploying such a system containing a large number of containers leads to an elaborated CI/CD (continuous integration and continuous delivery) pipeline, and the existence of a fully automated deployment pipeline is undeniable. To make such a pipeline, we require a container registry to be accessible from the both testing and production environment. A full path is necessary from the development environment (where we write the actual code for a new microservice) to the production environment, so that we can go through the pipeline to ensure that the whole system is working by adding a new microservices [27]. As we are porting a compiler from a monolithic approach to a microservice approach, it is quite possible to apply such a pipeline. Another benefit of the provided approach is that it will be possible to package a single module of a compiler with all its dependencies and configurations and transfer it to the release environment [28].

Additionally, by having access to the IR of the compiled code in each microservice, it is possible to conduct different analyses on the IRs during the compilation process or to even apply new optimization techniques. By the nature of the microservice pattern, we have enough freedom to involve different technologies and different programming languages in the compilation pipeline. By increasing the number of requests and the workload on the microservices, we benefit from the Canary deployment strategy and divide the network traffic among compiler microservices logically. Suppose we just added a microservice to the compilation chain and we need to test the workload or get feedback from users. In such a situation, by applying Canary deployment to the system, we can dedicate a percentage of the traffic to that specific service and prevent the whole system to going down in case of an exception in the new microservice. As a final benefit, it is possible to support several different compilers for different languages.

In a cloud environment, we need to somehow prepare an automatic scaling mechanism so that the cluster can grow horizontally and vertically based on demand. The provided approach is quite appropriate for such a mechanism, as services can be scaled up and down as they run independent pods and can increase the number of replications and spread them among the worker nodes.

However, we also see several limitations and hardships in migration. As current compilers have modules tightly coupled as one service, their separation and containerization is quite complex. We need to have a network layer in front of each module in the container, and each module has a different list of commands and flags. The front-line network layer needs to be aware of all of these and provide different TCP or HTTP APIs and then execute the most appropriate command to manage a service running in the background. A container with a module inside should be fully compatible with the compiler service running in the background in terms of microarchitecture design and limitations.

7. Deployment

The end-user has two possibilities to work with the service: CLI (Command line interface) and a Web application.

CLI (Command Line Interface)

To use the CLI, the user needs to get authenticated for the first usage by the system through a 256-SHA access token coupled with the user name and password. Users will get access to the compilation service only after system authentication. Listing 1 below provides a list of commands with appropriate flags that can be activated through the CLI.

Listing 1. CLI usage.

```
root@user: cloud-compiler -h
usage: cloud-compiler [OPTIONS] COMMAND
A multi-language cloud-based compiler

options:
-u      upload the code
-c      compile the code
-o      output
-v      version
-h      help
-w      watch
-o1     optimization level 1
-o2     optimization level 2
-03     optimization level 3
-r      download the report
-s      get the intermediate representation
-d      delete a code or a project
```

```
-a        activate the advisor
command
--auth        authenticate the user
--switch      switch between accounts
--plugin      activate a compiler plugin
--language    specify the desired language
--processor   type of the processor
--advisor     get the appropriate advice
--logout      logout from the account
```

As an example, a CLI code to run a C++ program with code advisor activated and level 3 optimization will look like Listing 2.

Listing 2. Uploading the code.

```
root@user: cloud-compiler -c source.cpp -o result -o3 -a -w -u -s
uploading the code...
##################################.....................60%
```

Figure 17 is a real deployment graph of the compiler as a cloud service on a Kubernetes cluster. The system as provided has a front-line gateway that takes the requests from the users and routes them to the back-end layer through the message broker. Then, all the compiler modules as well as new modules communicate through the same message broker. After compilation, the result of each pass can be transferred to the gateway and sent to the end-user as an HTTP response.

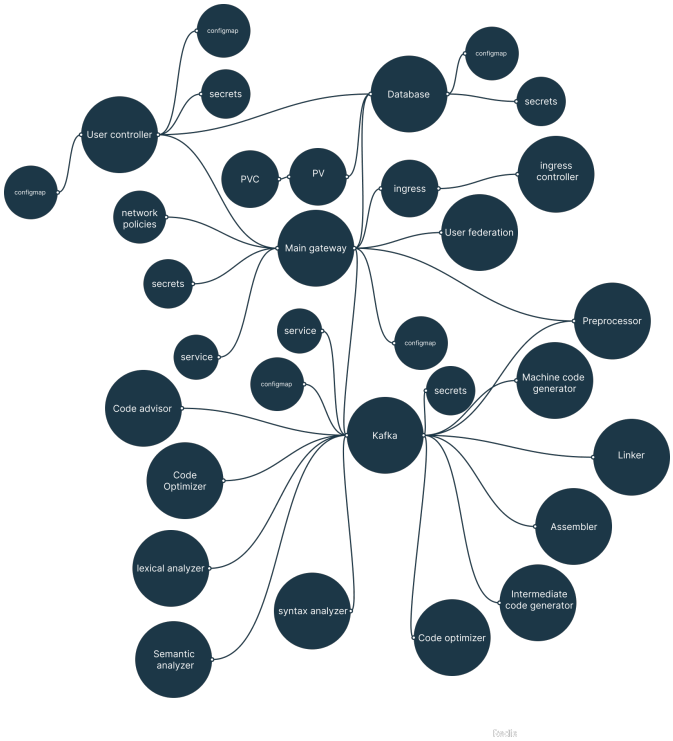

Figure 17. Deployment graph.

8. Experiments

A large part of overall cloud server performance comes from storage and read–write speed. In this research, we use the Fio benchmark, which is an I/O benchmarking and stress-test tool available on multiple platforms [29].

The specifications of the cloud service to run the experiments with one master node and two worker nodes are as follows:

- Processor: 2 × Intel Xeon Processor E5-2680 (20 M Cache, 2.70 GHz, 8.00 GT/s Intel(R) QPI) (16 Cores 32 Threads)
- Operating System: Ubuntu 20.0.4
- Storage: 2 × 300 GB 10K RPM SATA Hot Plug Hard Drive : Raid 1
- RAM: 32 GB RAM

In addition to the Fio benchmark, we also estimate the time for compiling an 8 × 8 matrix multiplication program written in C++ with LLVM/Clang compiler. We compare compilation time for a different number of worker nodes joined to the master node in the cluster. According to the experiment, we compile the code with level 3 optimization and code advisor activated. Then, we compare the time taken to send the code to the server and get the result. As provided in the plot, increasing the number of worker nodes decreases the compilation time accordingly.

The experiment is done with 10 replications per deployment in the cluster. The results are shown in Figure 18.

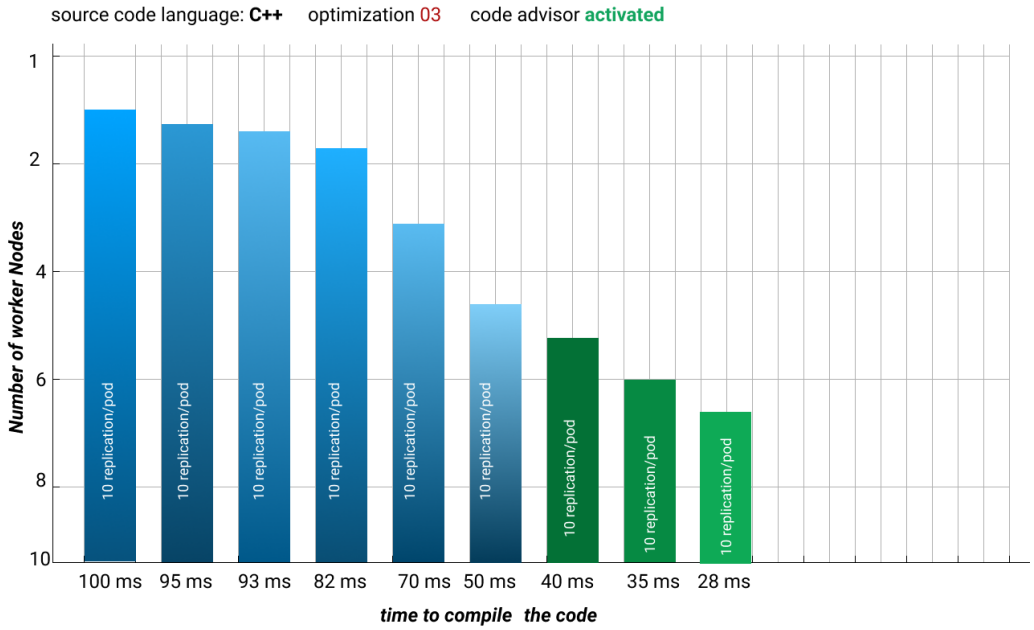

Figure 18. Compilation time.

Another experimental result is provided in Figure 19. By increasing the number of worker nodes joined to the cluster, the number of concurrent requests grows, and the number of concurrent users that the server can handle also increases. Server downtime decreases, and compilation speed increases.

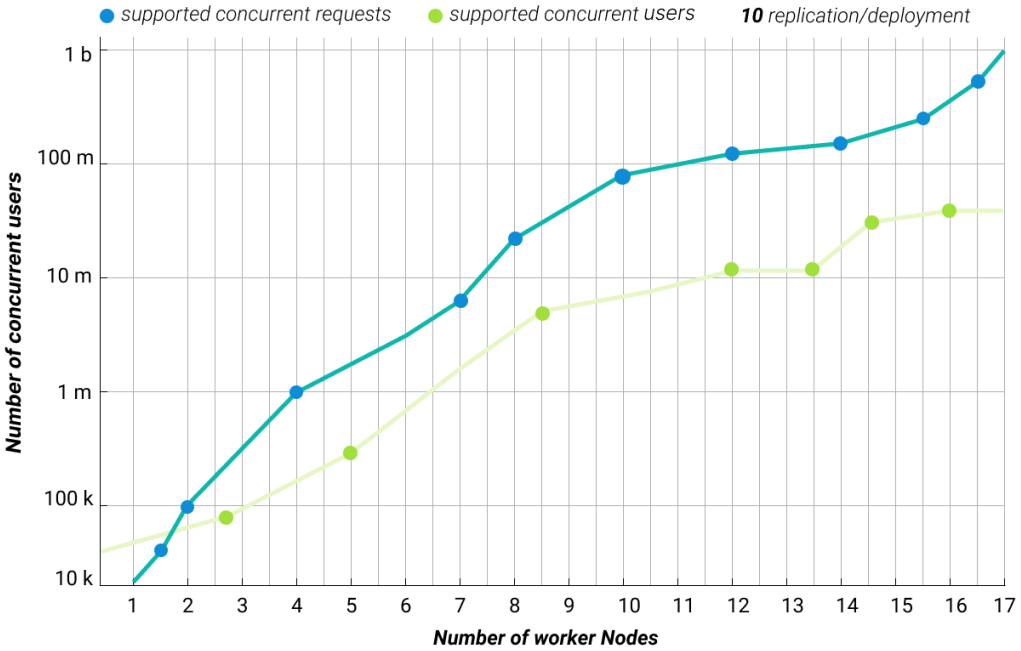

Figure 19. Concurrent users and requests.

9. Conclusions

Cloud computing is now quite ubiquitous, and the migration of standalone applications from local machines to cloud environments is undeniable. As the demand for Internet is increasing among users, it is necessary to provide centralized applications to users. Cloud computing in the software industry plays an important role in this modern technology. Providing applications as a collection of small units such as microservices makes auto-scaling easier and improves application accessibility.

In this paper, we have provided a new approach to deploying compilers on cloud infrastructure as a microservice-based architecture. According to the provided approach, we split the compiler from a monolithic service into several discrete, small services in different namespaces (front-end, middle-end, and back-end). Then, we proved we can have a service up and running with high availability on the cloud and provide a remote compiler for users to support many languages alongside other services such as an optimizer and an advisor. Additionally, experimental results showed much better compilation performance, as we were able to compile the program on more powerful hardware resources and generate the machine code in a short time and with a higher number of concurrent users.

To continue this research, we plan to add new optimization microservices to conduct the optimization at the IR level, and also implement an intelligent advisor that can produce useful advises for the programmer based on the CFG (control flow graph), generated IR, and machine code. With the data gathered from different code blocks, optimized algorithm implementation, and code-base designs, we plan to train a set of neural networks to recommend better optimized and higher-performance code to the programmer.

Author Contributions: Conceptualization, S.M.H. and A.A.P.; methodology, A.A.P.; software, S.M.H.; validation, S.M.H. and A.A.P.; formal analysis, A.A.P.; investigation, S.M.H.; resources, S.M.H. and A.A.P.; data curation, S.M.H.; writing—original draft preparation, S.M.H. and A.A.P.; writing—review and editing, A.A.P.; visualization, S.M.H.; supervision, A.A.P.; project administration, S.M.H.; funding acquisition, A.A.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Russian Science Foundation (RSF) project 22-21-00686, https://rscf.ru/en/project/22-21-00686 accessed on 9 August 2022.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CLI	command-line interface
SaaS	Software as a Service
TDD	Test-Driven Development
BDD	Behavior-Driven Development
EPT	Extended Page Tables
AST	Abstract Syntax Tree
GCC	GNU Compiler Collection
CI/CD	Continuous Integration and Continuous Delivery
CaaS	Containers as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
NaaS	Network as a Service
VM	Virtual Machine
LLVM	Low-Level Virtual Machine
UEC	Ubuntu Enterprise Cloud
CFG	Control Flow Graph

References

1. Mohammed, O.T.; Heidari, S.M.; Paznikov, A.A. Using OpenMP to Optimize Model Training Process in Machine Learning Algorithms. In Proceedings of the II International Conference on Neural Networks and Neurotechnologies (NeuroNT), Saint-Petersburg, Russia, 16 June 2021; pp. 21–24. [CrossRef]
2. De Lauretis, L. From Monolithic Architecture to Microservices Architecture. In Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 27–30 October 2019; pp. 93–96. [CrossRef]
3. Haptic. Available online: https://www.haptic.ro/ict-usage-in-enterprises-in-2018-12-of-enterprises-reported-analysing-big-data-and-4-used-3d-printing/ (accessed on 26 August 2022).
4. Chiplunkar, N.N.; Neelima, B.; Deepak. Multithreaded programming framework development for gcc infrastructure. In Proceedings of the 3rd International Conference on Computer Research and Development, Shanghai, China, 11–13 May 2011; pp. 54–57. [CrossRef]
5. Ghica, L.; Tapus, N. Optimized retargetable compiler for embedded processors—GCC vs. LLVM. In Proceedings of the IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, 3–5 September 2015; pp. 103–108. [CrossRef]
6. Waseem, M.; Liang, P. Microservices Architecture in DevOps. In Proceedings of the 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), Nanjing, China, 4–8 December 2017; pp. 13–14. [CrossRef]
7. Mulla, F.; Nair, S.; Chhabria, A. Cross Platform C Compiler. In Proceedings of the 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), Pune, India, 12–13 August 2016; pp. 1–4. [CrossRef]
8. Chebolu, N.A.B.; Sankar; Wankar, R. A novel scheme for Compiler Optimization Framework. In Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, 10–13 August 2015; pp. 983–986. [CrossRef]
9. Bokan, D.; Đukić, M.; Popović, M.; Četić, N. Adjustment of GCC compiler frontend for embedded processors. In Proceedings of the 22nd Telecommunications Forum Telfor (TELFOR), Belgrade, Serbia, 25–27 November 2014; pp. 983–986. [CrossRef]
10. Aamir, N.A.; Patil, S.; Navada, A.; Peshave, A.; Borole, V. Online C/C++ compiler using cloud computing. In Proceedings of the International Conference on Multimedia Technology, Hangzhou, China, 26–28 July 2011; pp. 3591–3594. [CrossRef]
11. Chandan, B.; Anirban, K.; Rana, D. SaaS Oriented Generic Cloud Compiler. *Procedia Technol.* **2013**, *10*, 253–261. [CrossRef]
12. Zhang, C.; Green, R.; Alam, M. Reliability and Utilization Evaluation of a Cloud Computing System Allowing Partial Failures. In Proceedings of the IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, 27 June–2 July 2014; pp. 936–937. [CrossRef]
13. Boyer, F.; Etchevers, X.; de plama N.; Tao, X. Poster: A Declarative Approach for Updating Distributed Microservices. In Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), New York, NY, USA, 27 May–3 June 2018; pp. 392–393.

14. Pabitha, M.; Selvakumar, T.; Punitha, D.S. An Effective C, C++, PHP, Perl, Ruby, Python Compiler using Cloud Computing. *Int. J. Comput. Appl.* **2013**, *69*, 20–25. [CrossRef]
15. Taherizadeh, S.; Grobelnik, M. Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *J. Adv. Eng. Softw.* **2020**, *140*, 102734. [CrossRef]
16. Park, C.; Han, M.; Lee, H.; Kim, S.W. Performance comparison of GCC and LLVM on the EISC processor. In Proceedings of the International Conference on Electronics, Information and Communications (ICEIC), Kota Kinabalu, Malaysia, 15–18 January 2014; pp. 1–2. [CrossRef]
17. Sill, A. The Design and Architecture of Microservices. *IEEE Cloud Comput.* **2016**, *3*, 76–80. [CrossRef]
18. Amanatullah, Y.; Lim, C.; Ipung, H.P.; Juliandri, A. Toward cloud computing reference architecture: Cloud service management perspective. In Proceedings of the International Conference on ICT for Smart Society, Jakarta, Indonesia, 13–14 Jun 2013; pp. 983–986. [CrossRef]
19. Haber, M.J.; Chappell, B.; Hills, C. *Cloud Computing*, 1st ed.; Apress: Berkeley, CA, USA, 2022; pp. 9–25.
20. Al-Debagy, O.; Martinek, P. A Comparative Review of Microservices and Monolithic Architectures. In Proceedings of the IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 21–22 November 2018; pp. 149–154. [CrossRef]
21. Petrasch, R. Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication. In Proceedings of the 14th International Joint Conference on Computer Science and Software Engineering, Nakhon Si Thammarat, Thailand, 12–14 July 2017; pp. 1–4. [CrossRef]
22. Kurnosov, M.; Paznikov, A. Efficiency analysis of decentralized grid scheduling with job migration and replication. In Proceedings of the the 7th International Conference on Ubiquitous Information Management and Communication, Kota Kinabalu, Malaysia, 17–19 January 2013; pp. 1–7. [CrossRef]
23. Sun, C.; Le, V.; Su, Z. Finding and Analyzing Compiler Warning Defects. In Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 203–213. [CrossRef]
24. Datta, A.; Paul, A.K. Online compiler as a cloud service. In Proceedings of the IEEE International Conference on Advanced Communications, Control and Computing Technologies, Ramanathapuram, India, 8–10 May 2014; pp. 149–154. [CrossRef]
25. Muddinagiri, R.; Ambavane, S.; Bayas, S. Self-Hosted Kubernetes: Deploying Docker Containers Locally With Minikube. In Proceedings of the IEEE International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET), Shegaon, India, 27–28 December 2019; pp. 239–243. [CrossRef]
26. Castro-Lopez, O.; Vega-Lopez, I.F. Multi-target Compiler for the Deployment of Machine Learning Models. In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, 16–20 February 2019; pp. 280–281. [CrossRef]
27. Górski, T. Towards Continuous Deployment for Blockchain. *J. Appl. Sci.* **2021**, *11*, 11745. [CrossRef]
28. Poniszewska-Marańda, A.; Czechowska, E. Kubernetes Cluster for Automating Software Production Environment. *Sensors* **2021**, *21*, 5. [CrossRef]
29. github. Available online: https://github.com/axboe/fio (accessed on 22 August 2022).