REPUBLIC OF TURKIYE
MANISA CELAL BAYAR UNIVERSITY
GRADUATE SCHOOL OF EDUCATION

MASTER THESIS
DEPARTMENT OF COMPUTER ENGINEERING
COMPUTER ENGINEERING PROGRAM

NARWHAL : A NEW VISUAL PROGRAMMING LANGUAGE TO MANAGE CLOUD INFRASTRUCTURE

Mehmet Emin KAYMAZ

Advisor
Doç. Dr. Övünç ÖZTÜRK

MANISA-2025

MEHMET
EMİN
KAYMAZ

NARWHAL : A NEW VISUAL PROGRAMMING LANGUAGE TO MANAGE CLOUD INFRASTRUCTURE

2025

TAAHHÜTNAME

Bu tezin Manisa Celal Bayar Üniversitesi Lisansüstü Eğitim Enstitüsü Bilgisayar Mühendisliği Ana Bilim Dalında akademik ve etik kurallara uygun olarak yazıldığını ve kullanılan tüm literatür bilgilerinin referans gösterilerek tezde yer aldığını, tamamen kendi çalışmam olduğunu, her alıntıya kaynak gösterdiğimi, tezin yazımında akademik ve etik kurallara aykırı herhangi bir yapay zeka ve program kullanmadığımı beyan ederim.

İmza

Mehmet Emin KAYMAZ

ÖZET

Yüksek Lisans Tezi

Mehmet Emin KAYMAZ
Manisa Celal Bayar Üniversitesi
Lisansüstü Eğitim Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Doç. Dr. Övünç ÖZTÜRK

Modern yazılım geliştirme süreçleri, karmaşık altyapılar ve mikroservis mimarilerinin yönetimini zorunlu kılmıştır. Bu durum, özellikle Infrastructure as Code (IaC) yaklaşımlarının popülerleşmesiyle birlikte, altyapı yönetiminde kod tabanlı çözümlerin sunduğu avantajların yanı sıra artan kompleksite ve anlaşılırlık sorunlarını da beraberinde getirmiştir. Narwhal adı verilen bu yeni görsel programlama dili, altyapı tasarımını interaktif ve görsel bir diagram üzerinden gerçekleştirme imkânı sunarak, IaC'nin mevcut zorluklarına yenilikçi bir çözüm getirmeyi amaçlamaktadır. Kullanıcılar, altyapı bileşenlerini ve yapılandırmalarını görsel olarak düzenleyerek Terraform HCL'ye dönüştürebilir ve altyapıyı doğrudan bu görsel şemadan derleyebilirler.

Narwhal'ın sunduğu görselleştirme yaklaşımı, altyapı tasarımlarının daha anlaşılır, paylaşılabilir ve kolay yönetilebilir olmasını sağlayarak, yazılım mimarisi erozyonunu azaltmayı hedeflemektedir. Yazılım mimarilerindeki hataların daha erken fark edilmesini sağlayan şematize yapı, aynı zamanda dokümantasyon maliyetlerini de önemli ölçüde düşürmektedir. Bu çalışmada, Narwhal'ın altyapı yönetimine sağladığı katkılar, yazılım mimarisi erozyonunu önleme konusundaki potansiyeli ve modern yazılım geliştirme süreçlerine etkileri detaylı olarak ele alınmaktadır. Narwhal, geleceğin altyapı yönetim araçları için hem bir vizyon hem de pratik bir çözüm önerisi sunmaktadır.

Anahtar Kelimeler : Görsel Programlama, Bulut, Terraform, Docker, Yazılım Mimari Erozyonu

2025, 95 sayfa

ABSTRACT

Master Thesis

Mehmet Emin KAYMAZ
Manisa Celal Bayar University
Graduate School of Education
Department of Computer Engineering

Advisor: Doç. Dr. Övünç ÖZTÜRK

Modern software development processes have made it necessary to manage complex infrastructures and microservice architectures. This situation, especially with the popularization of Infrastructure as Code (IaC) approaches, has brought about the advantages of code-based solutions in infrastructure management, as well as increasing complexity and comprehensibility problems. This new visual programming language, called Narwhal, aims to provide an innovative solution to the current challenges of IaC by offering the possibility of performing infrastructure design through an interactive and visual diagram. Users can visually edit infrastructure components and configurations, convert them to Terraform HCL, and compile the infrastructure directly from this visual diagram.

The visualization approach offered by Narwhal aims to reduce software architecture erosion by making infrastructure designs more understandable, shareable and easily manageable. The schematic structure, which allows for earlier detection of errors in software architectures, also significantly reduces documentation costs. In this study, Narwhal's contributions to infrastructure management, its potential to prevent software architecture erosion and its effects on modern software development processes are discussed in detail. Narwhal offers both a vision and a practical solution proposal for future infrastructure management tools.

Keywords : Visual Programming, Cloud, Terraform, Docker, Software Architecture Erosion

2025, 95 pages

PREFACE AND ACKNOWLEDGEMENTS

In this study, we have touched upon the details of the features of the visual programming structure we have developed, its areas of use, and examples of how it can be used. We hope that this visual programming environment will facilitate the storage and sharing of software infrastructures in a schematic form, thus contributing to the prevention of documentation management and software architecture erosion, which has become a very common problem in software systems today.

In this study, we first discuss software architectures and where these software architectures will evolve throughout the software development cycle. Then, we discuss the challenges that both the software team and the devops teams will face during this evolution process of software architectures and how mistakes made during the management of these challenges cause erosion in the software architecture.

Of course, we present solutions to these problems with Narwhal, along with usage scenarios and examples, as well as touching on the details of Narwhal's own architecture and working structure.

I would like to express my gratitude and respect to my university, which provided the necessary environment for this study, and to my advisor Doç. Dr. Övünç ÖZTÜRK, who supported me from the beginning to the end of my study and contributed with his knowledge and experience.

I would also like to express my respect and love to my mother, Nesrin KAYMAZ, and my father, Abdullah KAYMAZ, who have been by my side every moment and whose support I have felt every second during this challenging journey.

Mehmet Emin KAYMAZ

Manisa, 2025

LIST OF SYMBOLS

API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Delivery
CDKTF	Cloud Development Kit For Terraform
CI	Continuous Integration
CPU	Central Process Unit
HCL	Hashicorp Configuration Language
IaC	Infrastructure as Code
IoT	Internet of Things
JIT	Just In Time
JVM	Java Virtual Machine
MSIL	Microsoft Intermediate Language
SQL	Structured Query Language
UI	User Interface
VPC	Virtual Private Cloud

LIST OF FIGURES

		Page
Figure 1.	Narwhal Output Options	3
Figure 2.	Docker Composer Tool Diagram	5
Figure 3.	Dockstation User Interface	8
Figure 4.	Monolith Architecture	13
Figure 5.	Choreography-based Saga example	17
Figure 6.	Orchestration-based Saga example	18
Figure 7.	Narwhal Web Main Screen	68
Figure 8.	Narwhal Web Designer	69
Figure 9.	Narwhal Desktop Application Login Screen	70
Figure 10.	Narwhal Desktop Application Main Screen	71
Figure 11.	Generated C# .NET code	72
Figure 12.	Nodes in Json Model That Generated by Narwhal Web UI	73
Figure 13.	Edges in Json Model That Generated by Narwhal Web UI	74
Figure 14.	Generated HCL Code After Compilation	76
Figure 15.	A Simple Monolith App Example With Narwhal	78
Figure 16.	Private Docker Registry Example	79
Figure 17.	Running On Amazon Web Services	79

LIST OF TABLES

		Page
Table 1.	Docker Composer vs Narwhal Compare Results	6
Table 2.	Dockstation vs Narwhal	8
Table 3.	Programming Languages That Used Source-to-source Compilers	62

CONTENT

ÖZET I
ABSTRACT II
PREFACE AND ACKNOWLEDGEMENTS III
LIST OF SYMBOLS IV
LIST OF FIGURES V
LIST OF TABLES VI
CONTENT VII

CHAPTER ONE
INTRODUCTION 1
1.1. Related Works 2

CHAPTER TWO
AN OVERVIEW OF SOFTWARE ARCHITECTURES 11

2.1. The Role of Technical and Operational Factors in Architecture and Technology Selection 11
2.2. Monolith Architecture 13
2.3. Microservice Architecture 16
2.4. Scaling Architectures: The DevOps Challenge in High-Traffic Systems 19

CHAPTER THREE
SOFTWARE ARCHITECTURAL EROSION 23

CHAPTER FOUR
INFRASTRUCTURE AS CODE AND ITS CHALLENGES 30

4.1. Infrastructure as Code : Revolutionizing Modern Infrastructure Management 30
4.2. Complexity Issues of Infrastructure as Code 33
4.3. Terraform 34
4.3.1. Terraform and Its Contributions to Infrastructure as Code 36
4.4. Reducing Complexity in Infrastructure as Code with Visualization 40
4.5. Visual Programming Languages to Reduce Complexity in Infrastructure as Code 47

CHAPTER FIVE
SOURCE TO SOURCE COMPILERS 50

CHAPTER SIX
VISUAL PROGRAMMING 67

6.1. Purpose and Vision of the Narwhal Visual Programming Language 67
6.2. Architecture of Narwhal Visual Programming Language 68
6.2.1. Web User Interface of Narwhal Visual Programming Language 68
6.2.2. Desktop Application of Narwhal 70
6.2.3. Compiler Architecture of Narwhal Visual Programming Language 72

CHAPTER SEVEN
CAPABILITIES OF NARWHAL 78

7.1. Example Use-Cases of Narwhal 78
7.1.1. Monolith Architecture Example 78
7.1.2. Run Application From Private Registry 78
7.1.3. Running An Application On Amazon Web Services Example 79
7.2. Supported Functions by Narwhal 80
7.2.1. Docker Config 80
7.2.2. Docker Container 80
7.2.3. Docker Image 84
7.2.4. Docker Network 84
7.2.5. Docker Plugin 85
7.2.6. Docker Registry Image 86
7.2.7. Docker Secret 86
7.2.8. Docker Service 86
7.2.9. Docker Tag 87
7.2.10. Docker Volume 87

CHAPTER EIGHT
CONCLUSION 89

REFERENCES 91

1. INTRODUCTION

Nowadays, the needs of the software developed vary according to their own goals, for example, an application that can be accessed by several users may prefer a monolith architectural structure, while an e-commerce application will need a microservice architecture that can operate in a distributed manner on multiple servers. At the same time, the needs of the developed applications may change over time, which is why the developed architectures must be flexible. Otherwise, the software architecture will erode during the development or management processes of the software. [1][2]

Software systems with these different requirements also require software infrastructures with different configurations depending on the technology and architecture they use. Therefore, Infrastructure-as-Code and Containerization tools have become of great importance in this field.

The widespread use of Infrastructure-as-Code tools has made it easier to use architectures that were relatively difficult to implement. In particular, the use of microservices architecture has been paved by the use of Docker applications in a Docker container.[3] At the same time, these tools have also paved the way for comprehensive changes that software will experience during their life cycle, as they allow software infrastructures to be easily changed.

However, with the emergence of Infrastructure-as-Code tools such as Terraform or Ansible, in addition to software architectures, the configurations of the servers on which the software runs or is hosted have also been brought together under a single roof. This has made it necessary for developers to achieve a deeper level of knowledge in the areas of application infrastructure and architectural design.[4]

For this reason, many studies suggest using visual metaphors to both avoid architectural erosion in software and to abstract the complexity of Infrastructure-as-Code tools for architectural and software infrastructure development, which requires deep knowledge for software developers.[5]

In previous studies, Docker Compose configurations have been visualized by abstracting them from complexity, and it has been observed that these structures reduce

the error rate of developers.[5][6] Similarly, based on this and similar studies, we assume that implementing Docker functionality through a visual programming language and, moreover, ensuring that the disk, CPU, network or locations of servers in the Cloud, which can be managed through the Infrastructure-as-Code tool such as Terraform, which plays a key role in the Devops world, can be managed through the same visual programming language will further reduce the error rate of developers and facilitate the sharing of knowledge, software or server architectures.[7]

Thus, considering the advantages and conveniences provided by Narwhal, it can be seen that this situation can be used as an effective method to reduce or prevent software architectural erosion, which is frequently encountered in today's software architectures. In addition, these advantages and features will facilitate the understandability of software infrastructures, accelerate the infrastructure development processes of all small and large-scale software projects, and facilitate the maintenance and management of these systems.[8] In addition, since Narwhal can be compiled to Terraform's Hashicorp Configuration Language, it offers much more flexible features in the architectural field by abstracting the infrastructure as code structure one more turn. This will lead to the visualization of infrastructure as code tools.

1.1 Related Works

Narwhal provides great convenience in modern infrastructure design and management with its innovative approach offered as a visual programming language. The ability to compile node diagrams created in the design screen to Terraform’s HashiCorp Configuration Language (HCL) format is one of the main features that distinguishes Narwhal from other tools. In this way, even users with limited technical knowledge can visually create complex infrastructure designs and implement them using the power of Terraform. With the flexibility provided by Narwhal, users can edit these automatically created configuration files or use them as is and take direct action.

Narwhal also offers a great advantage for users who do not want to work with configuration files. Thanks to its visual design capabilities, users can design a system entirely from a node diagram and run this design directly without any manual intervention. For example, on a cloud platform, you can design server infrastructure,

network configurations or storage solutions only with Narwhal's visual tools and ensure that this system is installed. This reduces the operational workload of software development teams in particular, while also allowing non-technical teams to be involved in system design processes.

For users who want to create a similar structure with high-level programming languages, Narwhal also allows customization by converting the designed node diagram to high-level programming languages (C# . NET). For example, users can take code compatible with Terraform or other infrastructure management tools and perform more complex operations on this code. This flexibility allows users to easily adapt to their existing software development and infrastructure management processes. Thus, Narwhal offers both the power of visual design and the flexibility of low-level code manipulation.

Finally, another important advantage of Narwhal is that it can work in harmony with container-based platforms such as Docker. Users can manage not only the infrastructure but also container-based application deployments in the systems they design via Narwhal. For example, they can design a microservice infrastructure on a node diagram and perform the necessary network connections, database configurations and container arrangements through the same tool. This speeds up the end-to-end design and deployment of the system, while also minimizing the risk of errors. With these comprehensive capabilities, Narwhal offers a powerful solution to simplify and accelerate modern software development and infrastructure management processes.

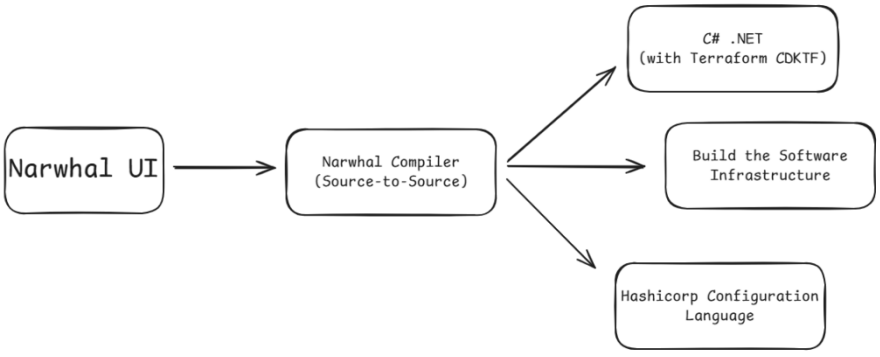

Figure 1. Narwhal Output Options

So, as seen in the figure 1, Narwhal offers the user 3 options.

- Compiling the diagram designed directly with Narwhal, and starting the software infrastructure designed in the diagram.
- Compiling the diagram designed with Narwhal, compiling the software infrastructure designed in the diagram to a C# .NET code using Terraform CDKTF, and starting the necessary software infrastructure using this C# .NET code.
- Compiling the diagram designed with Narwhal, compiling the software infrastructure designed in the diagram to Hashicorp Configuration Language, and starting the necessary software infrastructure using this Hashicorp Configuration Language.

Narwhal thus offers the user the option to be more dynamic. If necessary, the user will have the ability to bypass Narwhal by outputting the structure they designed with Narwhal as C# or Terraform code.

Narwhal aims to abstract the operations performed in the Infrastructure as Code and Containerization areas so that users can understand, change and share these structures more easily. Similarly, there are different studies in this area.

In one study, the automatic creation and use of Docker and Docker Compose configurations with a visual tool was discussed.[5]

If we look at the details of the study:

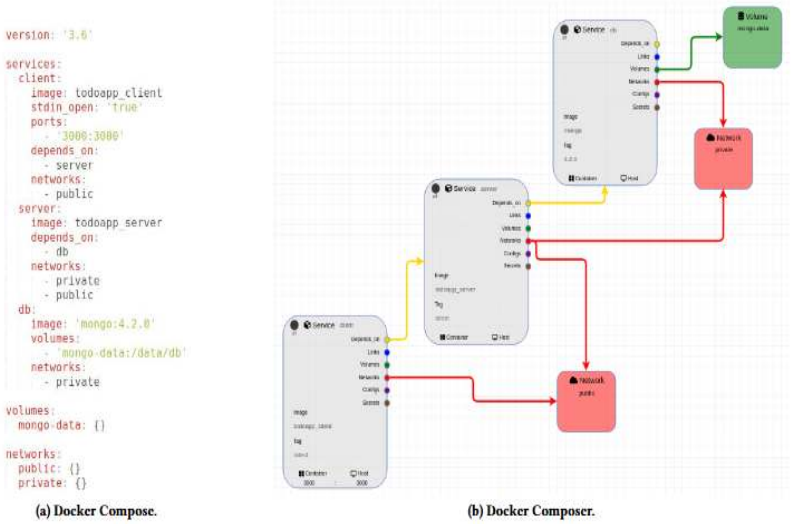

Figure 2. Docker Composer Tool Diagram

As can be seen in the figure, in the specified study, we see that the diagram was created in part (b) and in part (a), we see the Docker Compose configuration file that this diagram produces as output.[5]

Docker enables the relevant application to be launched by virtualizing applications in a container, isolating them from the operating system they are connected to, and hosting the libraries and tools required by the application in the container. Docker's approach is important because, since applications on the same server have similar needs in terms of resources and tools, errors that are difficult to detect and are caused by the versions of these tools when Docker is not used are quite common. In this respect, it is an indisputable fact that Docker and similar tools have an indispensable place in the software industry.

In this context, the study suggested that it would be more efficient to create a visual diagram of the configurations of such a frequently used docker container structure.

Similarly, Narwhal visual programming language supports the following docker features in the design made on the diagram:

- Docker Config
- Docker Container
- Docker Image
- Docker Network

- Docker Plugin
- Docker Registry Image
- Docker Secret
- Docker Service
- Docker Volume

In this respect, Narwhal visual programming language can support the features listed above since it targets Terraform, one of the Infrastructure as Code tools. In addition, Narwhal not only targets docker or docker compose configurations, but also includes the necessary tools to directly manage cloud infrastructure such as server-based, network, virtual private cloud, volume creation. In addition to this advantage, Narwhal also provides the user with the opportunity to produce code for certain purposes without being dependent on Narwhal and the diagrams it produces. Narwhal can provide the user with the infrastructure they design as a C# .NET code written using the Terraform CDKTF library or directly as Hashicorp Configuration Language. In addition, it can directly run the diagram without being dependent on these configurations and configure both the servers and the tools within the server.

A software infrastructure (docker-based) designed on the Narwhal diagram has the ability to directly perform the necessary installation on any server with ssh information.

However, if we compare on a docker basis, Narwhal supports more docker features compared to the work done.

Docker Feature	Docker Composer	Narwhal
Service	✓	✓
Network	✓	✓
Config	✓	✓
Secret	✓	✓
Volume	✓	✓
Tag	✓	✓
Plugin	✗	✓
Registry Image	✗	✓

Table 1. Docker Composer vs Narwhal Compare Results

If we leave aside Narwhal's server configuration etc. features and evaluate it only on a docker basis via Tablo.

Narwhal has the ability to integrate features that Docker Engine does not have by default into Docker Engine with plugin support, thanks to the Docker plugin feature it provides, and create user docker configurations accordingly.

Thanks to Registry Support, it allows you to manage the lifecycle of your existing docker images on the registry, and with all these features, it can automatically run this docker configuration and install the necessary tools according to the relevant configurations.

Narwhal visual programming language provides a structure that effectively optimizes the change management of a diagram created during the design phase. Changes made in the management of infrastructure elements are implemented by intervening only in the necessary components. In particular, in the event of a change made to a design, if no changes have been made to the server information or no manual operation has been performed on the servers, the system can perform the necessary updates on the target server based only on the changes made to the configuration.

This approach minimizes unnecessary operations in infrastructure management, saving both time and resources. Limiting changes made on servers to relevant components only increases the overall stability of the system and reduces the possibility of errors. In this way, the need to intervene in other components of the infrastructure is eliminated and configuration processes can be managed more dynamically.

This dynamic structure of Narwhal enables a more flexible and sustainable system design in infrastructure and software management. This flexibility provided by the visual programming language offers a significant advantage in terms of managing complexity and implementing changes more easily, especially in large-scale systems.

In conclusion, Narwhal’s innovative approach to infrastructure management strengthens the contribution of visual programming languages to the field of software engineering and makes change management processes more efficient. It is evaluated that such tools have the potential to create a significant paradigm shift in the field of software and infrastructure design.

Another study in this area can be seen as Dockstation. Dockstation can manage these services with a visual interface by launching the relevant Docker images as a Docker service via Docker Hub.

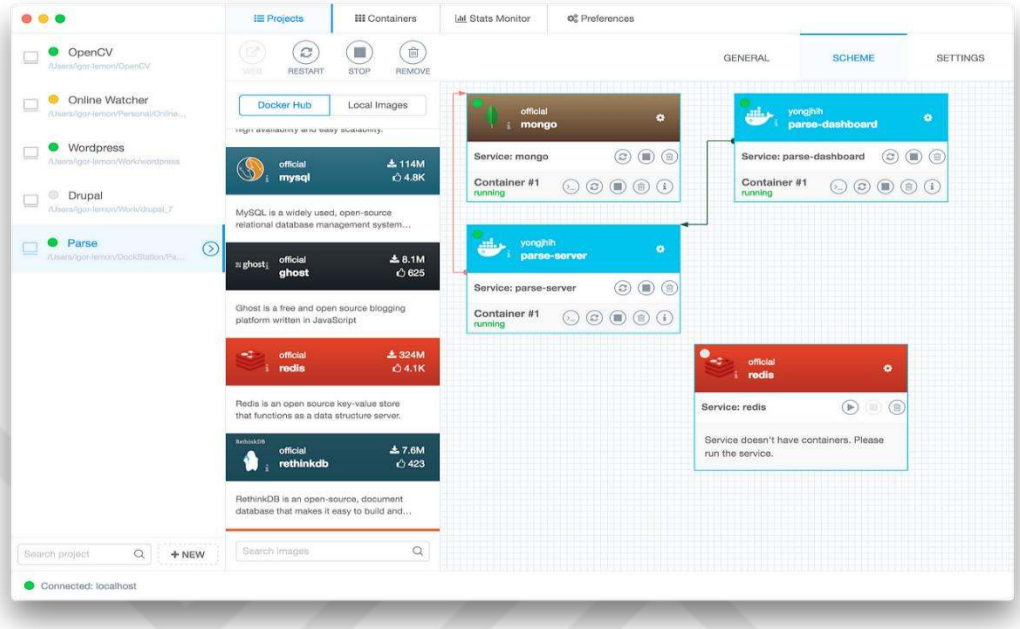

Figure 3. Dockstation User Interface

As seen in the figure, the Docker Image selected from Docker Hub in the left panel can be managed by going to the design screen on the right and managing the Service connections.

If we compare the Docker features offered:

Docker Feature	Dockstation	Narwhal
Service	✓	✓
Network	✗	✓
Config	✗	✓
Secret	✗	✓
Volume	✗	✓
Tag	✗	✓
Plugin	✗	✓
Registry Image	✗	✓

Table 2. Dockstation vs Narwhal

As can be seen in the table, Dockstation only supports the launch of images on Docker Hub as a service in a local or remote environment, but Narwhal has more capabilities in similar Docker functions and offers a more comprehensive environment to the user.

Narwhal stands out from other visualization tools in its field by offering an innovative approach to infrastructure management and DevOps processes, based on a powerful Infrastructure as Code (IaC) tool like Terraform. Combining the flexible and powerful features of Terraform with a visual programming paradigm, Narwhal offers more functionality to its users. This approach not only simplifies the management of infrastructure configurations, but also allows users to visually design infrastructure design and application processes, providing a more intuitive experience. Narwhal supports Terraform's HashiCorp Configuration Language (HCL) structure, enabling detailed definition and configuration of infrastructure components.

It enables more effective collaboration between infrastructure design, software development, and operations teams through a visual diagram. This level of abstraction provided by Narwhal makes software infrastructure configurations easier to share, manage, and modify. In this way, even team members without technical expertise can understand infrastructure designs and participate in the processes. At the same time, Narwhal's compatibility with the Terraform infrastructure allows existing configurations to be easily visualized and edited as needed. This creates an efficient information sharing and collaboration environment between software and DevOps teams.

These visualization capabilities provided by Narwhal reduce the complexity of infrastructure management, enabling changes to be implemented more quickly and reliably. Narwhal offers a flexible and dynamic structure to adapt to the rapidly changing needs of software infrastructures. This flexibility not only makes infrastructure designs more manageable, but also supports long-term scalability and sustainability goals. Based on the solid foundations of Terraform, Narwhal offers a significant innovation and contribution to modern software development and DevOps processes.

In this context, visual programming languages have become widespread in many software fields such as game development, Internet of Things (IoT), robotics and data science in recent years, offering developers an intuitive experience. The most fundamental advantage of such languages is that complex code blocks are represented in a visualized form and users can more easily participate in software development processes through these blocks. In this context, Narwhal offers a revolutionary approach to infrastructure management and operations by bringing the visual programming paradigm to DevOps processes. Narwhal not only visualizes DevOps processes, but also facilitates the management of these processes, the implementation of changes and the sharing of infrastructure designs. This provides a significant advantage, especially for teams working on large and complex infrastructures.

Narwhal's visual approach aims to increase efficiency and collaboration, especially in projects with high complexity in infrastructure management. While written definitions of infrastructure require technical expertise in traditional DevOps processes, these processes become more accessible with Narwhal. Designing infrastructure through visual diagrams allows different teams to more easily understand their impact on the infrastructure and to hold discussions on these designs. Narwhal enables team members with different skill levels to work on the same infrastructure, allowing business processes to accelerate and infrastructure decisions to be better managed.

Visualization of DevOps processes not only increases collaboration between teams, but also enables more transparent monitoring and management of the infrastructure. In this context, Narwhal offers an innovative solution by combining the flexibility and accessibility advantages provided by visual programming languages with DevOps processes. Visualization of infrastructure designs allows these designs to be reviewed more quickly and necessary changes to be implemented with fewer errors. This creates a significant advantage, especially for infrastructures that change and expand frequently, and places Narwhal in a unique position in modern DevOps approaches. It is anticipated that such tools, which integrate visual programming paradigms into DevOps processes, will play an even more important role in software development and infrastructure management processes in the future.

2. AN OVERVIEW OF SOFTWARE ARCHITECTURES

2.1 The Role of Technical and Operational Factors in Architecture and Technology Selection

In software projects, factors such as how and to what extent the resulting product will appeal to a user group, on which platforms it will be used, and the amount of traffic it must meet instantly should be determined in advance. The features that the software contains or will contain will have a direct impact on the software architecture. However, the effects of these features and project requirements are not limited to software architecture. The technologies to be used after the architecture decided according to the project plan also play a decisive role in system design.

For example, two systems that may be considered architecturally similar may be designed using different database management systems (e.g., Microsoft SQL Server or PostgreSQL), different messaging/queueing systems (e.g., RabbitMQ, ZeroMQ, Kafka), or different programming languages. However, the use of such tools depends not only on the technologies chosen, but also on how these technologies are configured at the server and infrastructure level.

For example, when you choose to run a database like PostgreSQL on AWS RDS, you need to consider the specific configuration options that AWS offers beyond a standard setup. Factors such as backup policies, auto-scaling settings, access permissions, and network configuration have a direct impact on performance and reliability. Similarly, when running a message queue system like RabbitMQ on Docker, it is critical to properly address details such as container resource limitations, network configuration, and log management.

In the process of Dockerizing these tools, processes such as using volumes for data persistence, configuring networks for containers to communicate with each other, and planning and managing high availability scenarios such as failover can become quite complex. For example, if PostgreSQL is run in Docker, database files must be stored correctly on the host machine and backed up securely when necessary. In addition, node clustering configurations in Docker containers are vital for the scalability and durability of the system for systems such as RabbitMQ.

Each configuration in these processes directly affects the performance and scalability of the technologies used. Therefore, expertise is required not only in software development but also in infrastructure management. As the variety of tools and infrastructure options increases, the configuration and management processes of the system become more complex, which puts significant pressure on the competence of the team developing the project. An incorrect or incomplete configuration can lead to problems that are difficult to fix in the next stages.

Therefore, the software architecture and technology selection process should consider not only the goals of the software, but also the management and maintenance of the infrastructure. Non-technical factors such as the size of the company developing the project, the capacity of the DevOps team, or budgetary constraints are important factors that can affect these decisions. For example, if you do not have a large DevOps team, building your software infrastructure in the cloud may be a more attractive option. On the other hand, if you have sufficient resources to manage your own infrastructure, alternative solutions may be preferred.

The decisions to be made here directly affect the technologies used; because most cloud platforms present the selected products to the user by virtualizing them. For example, there may be restrictions such as query sending limits for a database on the Microsoft Azure platform. In addition, the performance of a virtualized database may differ from a directly managed physical database. If such limits and configuration requirements are not taken into account, the performance and reliability of the system may be negatively affected.

Therefore, architecture and technology choices in software projects should be made not only in line with technical requirements, but also considering the long-term manageability of infrastructure and maintenance processes. The complexity of these processes is directly proportional to the knowledge and experience of the team, and each of these elements should be carefully planned for a successful project.

However, even if all these plans, operations and preferences are done correctly, it will not eliminate all the problems in software projects. Software projects have a dynamic life cycle and in this cycle, they constantly evolve even after the project is published. A published software project will have to be updated depending on factors such as user feedback, changing business requirements or sectoral innovations. During

this process, the software may gain new features, optimize its existing features or encounter adjustments to adapt to changes in the libraries, packages and technologies it depends on.[9]

For example, even if a software project was initially designed with a specific database system, it may be necessary to switch to a different database type as the number of users increases or as the system becomes available in different geographic regions. Similarly, third-party libraries that the software depends on may become outdated or become obsolete over time. This means that the code used in the project may need to be updated, refactored, or changed. All of these changes require a development process that ensures the continuity of the project.

During this process, there may be changes not only in the software but also in the team that develops the software. Developers may leave the team, new developers may join the team, or they may be directed to different projects within the company. In such changes, it is critical that the knowledge and experience (know-how) regarding the software infrastructure is effectively transferred to the new developers. If this knowledge transfer is not successfully carried out, the sustainability of the software project may be at risk.[2]

2.2 Monolith Architecture

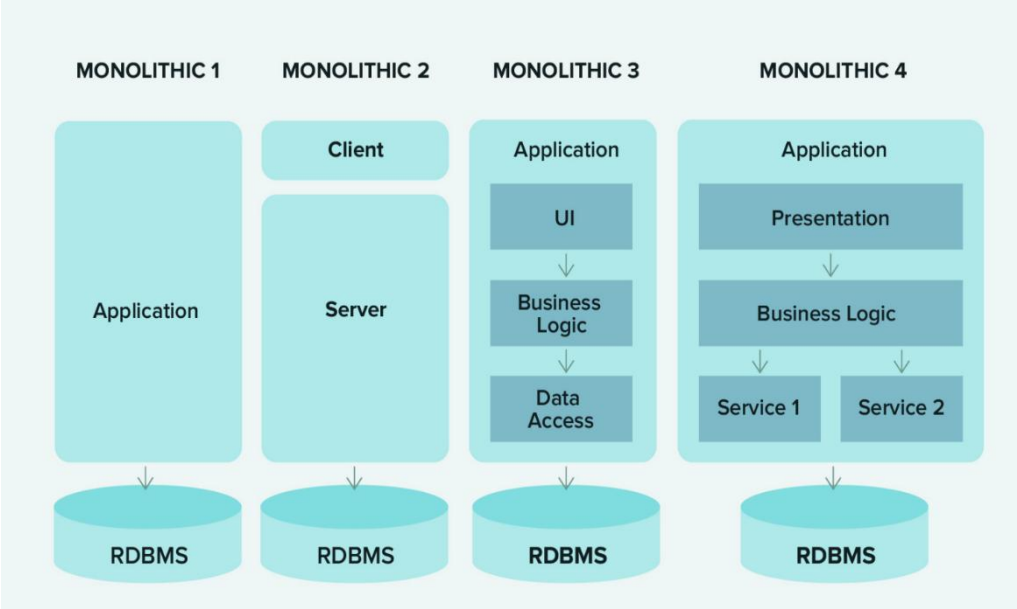

Figure 4. Monolith Architecture

Monolithic architecture describes a tightly coupled structure in software projects, usually consisting of a single user interface, service endpoint, and database.[10] This structure is usually a structure where all components are developed in a single programming language and the entire system runs on a single database. One of the advantages of a monolithic structure is that it does not require complex processes such as data transfer, synchronization, and distributed transaction management, as all components are located in a central system. As a result, devops processes, CI/CD management, and the overall software development process for such projects can be shorter and simpler. This reduces software development time and requires less technical effort.

However, when a monolithic architecture encounters high user traffic, the structure may have great difficulties in handling this traffic. The main problem of a monolithic structure is that all components are grouped under a single API. This requires the API to manage all traffic alone. Under high user traffic, a single API can affect the performance of the entire system and cannot properly handle the traffic. Even if more API instances are created for this, this solution will not always be sufficient because APIs can be inefficient due to traffic not being distributed equally to each service endpoint. Less used endpoints can consume unnecessary resources, which prevents the system from operating efficiently.

This centralized management via a single API is another important factor that limits the scalability of the software. Scalability refers to the ability of a software to cope with increasing loads. Monolithic structures can have difficulty providing this scalability for high-traffic applications. Increasing the load on the API is not only done with more server resources, but also with more processing power. However, each server instance only manages one API, so the load on the database will increase. As a result, increasing the number of APIs can further increase the load on the database in the system, which leads to a serious throttling of database performance.

Deploying across multiple servers can help alleviate these challenges in a monolithic architecture. However, this solution also brings with it some significant problems. Because monolithic architectures are usually based on a single database, and this database can have difficulty managing requests from multiple servers. Load balancing can be done between servers, but having each server access the database at the same time increases the load on the database. In this case, a bottleneck occurs as

the database encounters more requests, which prevents the software from working efficiently.

One of the biggest disadvantages of the monolithic structure is the bottlenecks that occur in the later stages of the project due to high user traffic and database density.[11] As a result, the software project requires more server and processing resources. However, due to the limitations of the monolithic structure, these resources are difficult to provide, which leads to scalability problems in the project. Such problems pose a serious risk to the sustainability of the software, because if resources are not managed effectively, system performance can deteriorate rapidly.

Finally, in order for monolithic structures to overcome such problems, a transition to more flexible and distributed architectures may be necessary. However, this transition requires the restructuring of the existing structure and a large-scale redesign process. Such a restructuring process can lead to high costs and loss of time.[10] Therefore, although monolithic structures may initially seem like structures that can be developed more easily and quickly, in growing and popular projects, this structure becoming unmanageable can threaten the long-term sustainability of the software.

In other words, the fact that monolith architecture can be developed easily and quickly is a good advantage, but it requires the software project to develop and evolve without errors in terms of architecture over time.[12] A software project designed with a monolith architecture should be able to dynamically transition to a microservice architecture when exposed to high traffic in the future.[12] In this respect, monolith architectures should be modular when designed, hybrid approaches should be considered in database designs, and it should be prepared for dynamic changes when necessary.[13] Although monolith architecture is less preferred today with the increasing popularity of microservices, it can still be a valid solution in small projects.[14] Modularizing the monolith structure can facilitate the transition to microservices in the future. This transition can be especially useful as the requirements of the application increase. As a result, while monolith architecture offers an efficient approach under certain conditions, more flexible solutions such as microservices can be preferred in large projects.

2.3 Microservice Architecture

Microservices architecture is a modern architectural design approach that is increasingly preferred, especially in medium and large-scale software projects.[8] This architecture aims to provide flexibility in software development and management processes by separating applications into smaller, independent and functional units.[15] Microservices architecture includes various application methods and design patterns; this allows software teams to make project-specific customizations.

The basic principle of microservice architecture is to divide all the features of an application into multiple services (APIs) that can work independently of each other and form a holistic system when combined. One of the biggest advantages of this approach is that instead of concentrating heavy traffic on a single API or server, the load is distributed across multiple services and servers. This distribution is usually done through a load balancer and allows the system to respond to more user requests instantly.

In addition, microservice architecture provides great benefits in terms of fault isolation. In a single monolithic application, an error in one section can affect the entire system, whereas with the microservice approach, errors in one or more services do not affect the operation of other services. This prevents general service interruptions and ensures that the system operates more resiliently and without interruption.

As a result, microservices architecture has become an indispensable part of modern software development processes, offering high performance and reliability in large-scale projects, thanks to its advantages such as flexibility, scalability and fault tolerance.

In this context, solutions such as Saga Pattern come into play to further enhance the flexibility and fault tolerance advantages offered by microservices architecture. Saga Pattern is a model developed to manage long-running transactions involving multiple services in distributed systems. Used to ensure transaction integrity between microservices, this approach breaks transactions into a series of small and independent steps.

There are two common implementations of the Saga Pattern: Choreography-based Saga and Orchestration-based Saga. The Choreography-based Saga method

allows each service to trigger an event to start the next process without the need for a central coordinator. This method allows processes to be organized dynamically by creating an event-based communication infrastructure between microservices. We will now examine how this approach works and its application in a real-life scenario.

For example, **Choreography-based Saga**,

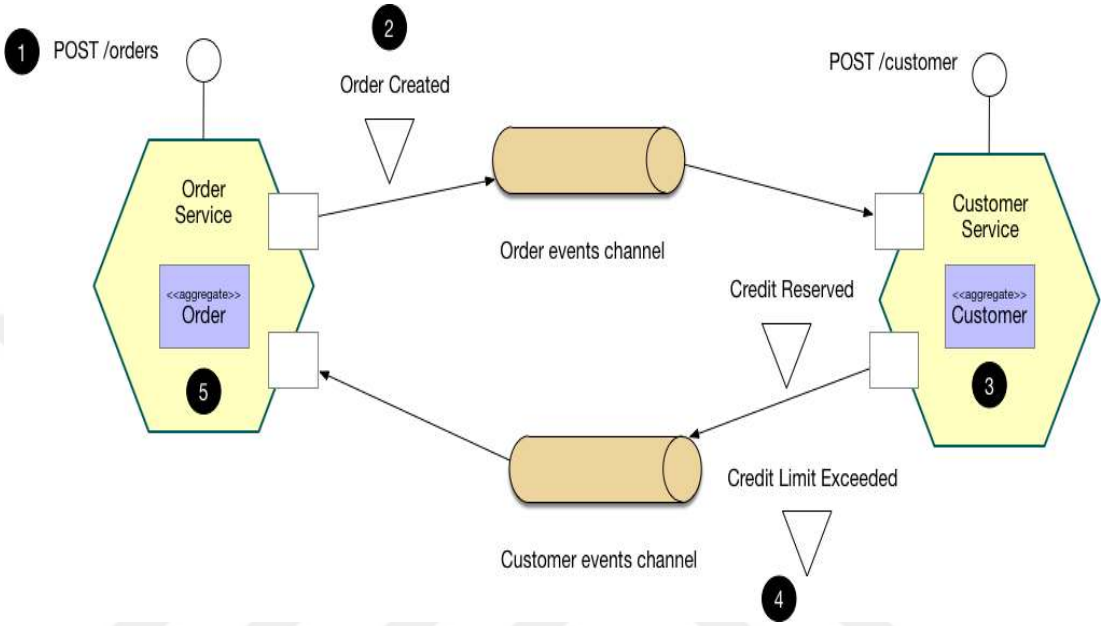

Figure 5. Choreography-based Saga Example [16]

In the model shown in Figure 5, there are two services named Order and Customer. These services communicate with each other using a message broker and there is a database for each service.[16]

The basic flow can be expressed as follows:

1- A request comes to the Order Service

2- The Order Service records the relevant order as pending in its own database

3- The Order Service sends an OrderCreated event to the queue.

4- The Customer Service, which receives the OrderCreated event, sends an event to the Customer queue if the customer's balance is sufficient.

5- If the Order Service receives a sufficient balance response, it actively marks the status of the order it has recorded as pending.

For example, **Orchestration-based Saga**,

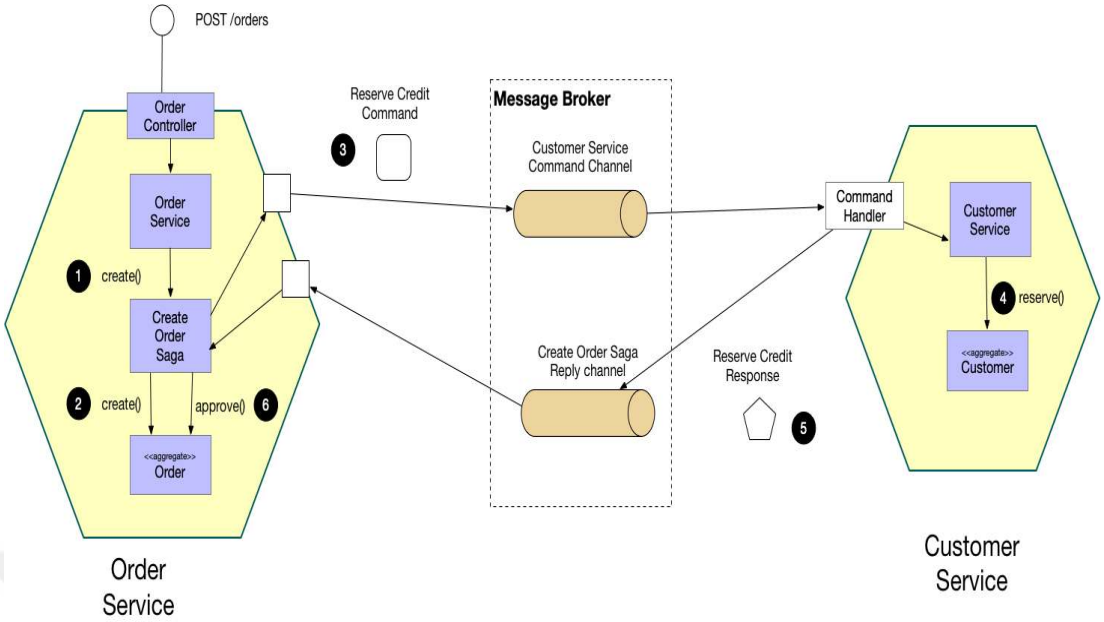

Figure 6. Orchestration-based Saga Example [16]

The basic flow can be expressed as follows:

1- The Order Service receives the POST /orders request and starts the Create Order saga orchestrator.

2- The Saga orchestrator creates an Order with the PENDING status.

3- It then sends a Credit Reservation command to the Customer Service.

4- The Customer Service attempts to reserve the credit.

5- It then sends a response message indicating the result of the transaction.

6- The Saga orchestrator either approves or rejects the Order.

Microservice architecture is a structure that can be customized according to different requirements because it allows each service to work independently. In this context, while a service may need a NoSQL-based database, a SQL-based solution may be more suitable for another service.[17] Similarly, services can be written in different programming languages and different messaging infrastructures such as RabbitMQ and Kafka can be preferred to ensure their communication.

In an environment where different services are developed by different software teams, each service should have its own CI/CD (Continuous Integration and

Continuous Delivery) process and DevOps practices. This increases independence between teams, allowing each team to define their own tools and processes according to their needs. However, this independence also brings with it a more complex management structure.[18][19] Separate testing processes, automation systems, and deployment pipelines may need to be created for each service.[20]

Since a real application does not consist of just a few services, but often dozens or even hundreds of services, there is a great deal of diversity in the tools and configurations used throughout the system. The use of different technologies and tools depending on the technical requirements and functionality of the services can make the microservices ecosystem quite complex.[21][20] This complexity requires extensive coordination and detailed documentation, not only in the development and deployment processes, but also in the maintenance and monitoring phases.

In such an environment, modern tools play a critical role in ensuring that services operate independently while maintaining the overall integrity of the system. Tools such as Kubernetes and Docker Swarm for container orchestration make it easier to deploy, scale, and manage containers. In addition, tools such as Terraform and Ansible for automation and configuration of the infrastructure make system management more efficient. These tools contribute to the scalability, resilience, and sustainability of the system by making complexity manageable, albeit less. Docker is the cornerstone of this ecosystem as a fundamental tool for building and running containers.

2.4 Scaling Architectures: The DevOps Challenge in High-Traffic Systems

The software architecture to be used in today's software projects and the way they are implemented are very important. Applications that will be used by millions of people or smaller projects must be developed with a software architecture that is unique and open to development.

If we generalize from the software architectures frequently used today, two main software architectures emerge. These are monolith and microservice architectures.

While a software project can initially serve a limited number of users, it must reach the capacity to serve millions of people over time if the number of users increases. This requirement may necessitate a transition to a microservice structure, especially if the software is initially developed with a monolithic structure.[22] Therefore, it is very important that a software initially developed as a monolith is divided into microservices over time and transformed into a more efficient and scalable structure. This raises the question of how to manage the architectural changes, technological requirements and infrastructure needs encountered during the development process of software projects.[23][8]

The development and maintenance of software projects is a process that often requires changes in the software infrastructure, tools used, and CI/CD processes.[23] These changes are made in order to respond to the increasing user demands along with the evolution of the software. However, such changes affect not only the functional structure of the software, but also the server infrastructure and the configuration of the technologies used. Therefore, the growth and evolution of a software project not only meets the functional requirements of the application, but also has significant effects on the infrastructure and operational processes.

This constant change and evolution process poses major challenges in terms of management, especially in the DevOps field. DevOps processes aim to efficiently manage software infrastructure changes, deployment processes, and resource management.[24] However, when a software starts to be divided into microservices, each service may need a separate database, which can make infrastructure requirements more complex.[25][26] In addition, these microservices may need to be distributed across multiple servers to handle high traffic. Managing such infrastructure requirements is one of the main challenges faced by DevOps engineers.

Rapid change processes in particular can be an obstacle to a successful scalability process.[24] In cases where sufficient documentation is not provided for rapidly changing infrastructure and software structures, difficulties may be experienced in managing the system in the long term.[27] This situation can lead to infrastructure errors and administrative problems, especially during the software growth process. In this context, careful documentation and management of every change made in software projects is critical for a sustainable and efficient scalable infrastructure.

Therefore, the evolution of software projects requires effective management of DevOps processes. Every change in infrastructure and software architecture requires proper planning and coordination. Successful management of such changes is a fundamental element for the software to serve a wider user base and operate with high efficiency over time. The ability of DevOps engineers to effectively manage such change processes plays a critical role in the success of software projects.

In this context, providing fast, reliable and scalable solutions in modern software projects has become one of the key elements of gaining a competitive advantage. In this context, DevOps aims to increase collaboration between teams, accelerate the software lifecycle and ensure error-free, continuous delivery in the production environment by integrating software development (Development) and operations (Operations) processes.[28] DevOps is not only a methodology; it is also a comprehensive cultural transformation process supported by modern techniques such as automation tools, cloud technologies and continuous integration/continuous delivery (CI/CD).

However, the number of tools to be used in today's software systems and the number of cloud system providers to be preferred have increased, and each of them requires different structural features and therefore different customized configuration settings, making it difficult to manage DevOps processes.

Of course, not only the variety of configurations and cloud solutions, but also the high traffic of today's software projects, the complex software architecture solutions that are used to meet these, and the project-based isolation of software teams make it difficult to manage DevOps processes.[29] DevOps processes have to deal with difficult problems based on technical or non-technical reasons, such as installing and configuring servers for complex software architectures, updating them during the software lifecycle, and documenting the work done. [29]

The ability of software projects to manage challenging conditions such as high traffic requires not only software architecture but also successful management of DevOps processes. This process requires continuous infrastructure changes, network configurations and resource management to optimize the scalability and performance of the software. In this context, DevOps engineers are responsible for managing continuous integration, deployment and monitoring processes by integrating

infrastructure with software development processes.[28][30] However, meeting such complex infrastructure and operational requirements of software projects requires a high degree of planning and coordination. The Narwhal visual programming language we developed makes a significant contribution to managing these complex processes. Narwhal makes DevOps and infrastructure processes more efficient by adding a visual abstraction layer. While allowing users to design infrastructure on visual diagrams, it also allows them to compile these designs into more technical and structured languages such as HashiCorp Configuration Language (HCL). In this way, users can design and implement their systems more easily, facilitate the infrastructure management required in DevOps processes and manage them with fewer errors.

3. SOFTWARE ARCHITECTURAL EROSION

If we need to define erosion in software architectures, this situation can be expressed as the deviation of a software project from the goals and standards determined in the initial phase over time.[31] Software projects are usually designed to meet a certain traffic load, and the programming languages, technologies and architectural approaches to be used are carefully selected in accordance with the requirements and goals of the project. Similarly, DevOps processes, continuous integration and continuous delivery (CI/CD) approaches, server configurations and other infrastructure elements are designed and implemented to meet these requirements of the project.[32] However, software projects have a dynamic structure and are subject to many changes throughout their life cycle. These changes may cause deviations from the initial plans.

Technical and non-technical decisions made during the software development process can negatively impact the initial architectural integrity and performance expectations of the system.[1] For example, deviations from initially planned design principles can lead to accumulation of technical debt; lack of documentation can undermine knowledge sharing within teams; or temporary solutions to meet short-term business needs can jeopardize the long-term sustainability of the system. All of these factors can cause the software to fall short of the initial standards and performance expectations.

Architectural erosion is not limited to technical decisions; also, situations such as misdirection by decision makers who do not have technical knowledge or focusing on short-term goals can accelerate this process.[1] As a result, deviation from the initially determined architectural and technical goals causes the software to fail to deliver the output as initially planned, both functionally and technically. This makes the software difficult to maintain, increases costs, and complicates future developments. This type of process is defined as erosion in software architectures and emerges as a significant threat affecting the long-term success of software projects.

There are many reasons why software architectures cannot be implemented as planned, here are a few examples:

- Technical incompetence of software developers
- Bad architectural decisions based on insufficient technical knowledge
- High turnover within software companies
- Lack of documentation in developments
- Impact of non-technical decisions on software architecture

If we touch upon these items, for example, the technical inadequacy of software developers, the lack of sufficient, proper algorithm and technical knowledge during the implementation phase of the planned architecture, and the fact that the written code does not give the desired performance may affect the overall performance of the application. The fact that these errors are made in an area that can be considered the core of the application will also make it difficult to compensate because this area will be used by perhaps hundreds of classes and methods. The time spent to compensate for these errors can be quite high, and the desired result in the compensation process may not give the expected output due to technical inadequacy.

Similarly, decisions taken by software architects with insufficient technical knowledge or taken at the very beginning may cause problems in the application over time. These problems are the most difficult to compensate for because they directly affect the software architecture and not only that, they also include the technical tools and their configurations to be used in the architecture, and they are broad in scope and require time and effort to solve. For example, let's assume that a monolithic application architecture is planned for a software project, the architecture should be designed modularly so that a single API and a single database do not cause a bottleneck in the application when it starts to receive high traffic in the future, or the tools to be used in another sample application should be selected according to the need. For example, if an Object-Relational Mapping tool is not used for a large project, but instead an attempt is made to manage workflows directly by calling SQL codes or through stored procedures, thousands of lines of code may need to be revised or rewritten for a change to be made in the database in the future. Many examples like this can be listed, but the important point here is that architectural errors caused by technical inadequacy cannot be compensated for in the future. Apart from the lack of technical knowledge, other problems can also directly or indirectly cause erosion in software architectures. One of the most important of these is the shortening of the starting and ending intervals of

employees in technical roles within software companies. When a team member working on a software project leaves work, he takes his technical knowledge with him. Of course, a new person will come in his place and transfer his knowledge within the process, but if there is an incomplete transfer due to issues that are skipped or time is not found during these transfer processes, there may be a loss of technical knowledge. However, if there is a rapid turnover of employees within the company, the amount of lost technical knowledge increases. This situation may bring about short-term or long-term problems that may arise from knowingly or unknowingly skipping certain critical points in the developments made during the software life cycle. This may also cause an architectural erosion in the relevant project.

Dynamics such as employee turnover in software companies can create significant challenges in terms of project sustainability. In particular, changes in team members can make it difficult for new individuals to understand existing systems and negatively affect their productivity. One of the most effective ways to minimize such effects is to create comprehensive and well-prepared technical documentation. Technical documentation provides a guide for both current and future team members by documenting the architectural structure of a software system, the operation of the code, and design decisions in detail. However, in the software development process, sufficient time and resources are often not allocated to documentation work. Especially in cases where the project needs to progress rapidly, keeping up with the pace of developments in the architecture or code base can make comprehensive documentation processes quite difficult.

Failure to provide timely and accurate documentation may have negative consequences that negatively affect the architectural integrity of the project in the long term. Incomplete or scattered knowledge of the system within the team may lead to misinterpretations and uncertainties in decision-making processes. In addition, inadequate documentation of changes made to the architecture may lead to a failure to understand why and how these changes were made in the future. This situation paves the way for team members to be misguided when adding new features or improving the existing system, thus making architectural erosion inevitable. As a result, neglecting or delaying technical documentation not only leads to loss of knowledge in software projects, but also creates serious negative effects on the sustainability and scalability of the system, thus accelerating architectural erosion.

The impact of technical decisions and the technical team on the future of software architecture can be observed directly and clearly. It is an undeniable fact that technical decisions taken in software projects are the basic elements that determine the functionality, performance and sustainability of the system. However, research and field studies show that the biggest reason for the erosion observed in software architectures is not technical decisions, but the effects of non-technical decisions. These decisions usually have indirect but deep effects on software development processes, and these effects can lead to the weakening of architectural integrity in the long run.

For example, in order to reduce financial costs, some organizations may prefer to use credits offered by cloud service providers. These credits initially offer an attractive solution to cover the technical infrastructure required by the software project. However, if the tools used or the cloud provider are considered to be changed close to the end of the credit period, this may have significant consequences affecting both the technical development processes and architectural decisions. In particular, situations such as waiving some technical features accessed within the scope of the credit can seriously affect the performance and sustainability of the project. Such decisions can both increase technical development costs and cause extensive architectural changes in the system.

For example, migrating from one database system to another in order to reduce costs is a complex and costly process in many ways. Such a decision involves extensive changes that will affect not only data migration but also server installation configurations, automated migration structures, and overall system architecture. Moreover, such a migration may require extensive refactoring of the codebase, which can create additional burden on the technical team.

Making such decisions without sufficient thought or coordination with the technical team can have irreversible effects on the architectural integrity and weaken the basic building blocks of the software architecture. Therefore, a strategic approach should be adopted in making non-technical decisions and decision-making processes should be designed with the long-term sustainability of the software in mind. Otherwise, these decisions can become critical factors that directly pave the way for architectural erosion and seriously jeopardize the future success of software projects.

From another perspective, the software development process often focuses on completing tasks quickly.[33] This focus can lead to the neglect of important but time-consuming tasks such as technical documentation. Technical documentation records the operation of the software system, design decisions, and architectural structure, providing reference information that will be needed in the future. However, due to time pressure or lack of resources during the development process, such tasks are either left incomplete or not done at all.[33][34] Lack of documentation can lead to serious problems in the later stages of software development. For example, when a new team member joins the project, a bug in the existing system needs to be fixed, or a feature needs to be added, the lack of sufficient documentation can lengthen the process and increase complexity. More importantly, deficiencies in documentation can lead to poor knowledge sharing within the team and, over time, to forgetting the basic principles in the design of the system. This leads to unconscious violations of architectural decisions and the system facing technical debt accumulation in the long term. As a result, neglecting documentation becomes an important factor that indirectly accelerates architectural erosion in software systems.

Unlike technical decisions, it is more common for non-technical decisions to cause architectural erosion. The main reason for this is that some of the decisions made in projects are shaped by managers or decision makers who do not have technical knowledge. These people usually prioritize achieving short-term goals or meeting business needs because they do not fully understand the complexities or long-term effects of the software development process. As a result, quick and temporary solutions may be preferred over long-term sustainable solutions suggested by the technical team. Losing the freedom and decision-making authority required for technical teams to correctly direct the project can cause technical debt to accumulate in projects and weaken architectural integrity. Such situations can seriously complicate future maintenance, development and scalability of the software. Therefore, the effects of non-technical decisions on architecture are a critical issue that must be carefully addressed in software projects.

Narwhal is designed as an innovative visual programming language that redefines software architecture processes. This language aims to provide solutions to the challenges that are often encountered in software projects, such as architectural complexity, lack of documentation, and manual configuration. The main goal of

Narwhal is to enable more understandable, faster, and more reliable design and implementation of software architectures by adopting a diagram-based design approach. Visualized architectural diagrams on diagrams not only greatly reduce the need for documentation, but also allow these diagrams to be compiled directly into server configurations. This automates the manual infrastructure setup processes that are often time-consuming and error-prone in software projects.

Narwhal automatically creates and configures servers on relevant cloud providers based on the designed schema. Messaging systems (such as RabbitMQ), databases (such as MSSQL, PostgreSQL) and other technical tools that will run on the servers are also configured with their necessary settings during this process. In addition, technical details such as network settings required for communication between servers are also organized by Narwhal. In this way, technical errors that may arise from manual interventions are minimized and infrastructure processes are accelerated. Thanks to its diagram-based approach, Narwhal provides a visual representation instead of hundreds of lines of code, allowing software architectures to be understood and easily shared within the team. The fact that even non-technical team members understand the diagrams increases communication and collaboration in software projects.

One of the most striking features of Narwhal is its potential to reduce the problem of architectural erosion, which is frequently encountered in software projects. Architectural erosion can be defined as the deviation of architecture from the initial design goals due to unplanned changes in projects, lack of documentation, or non-technical decisions. Narwhal reduces the impact of such deviations by providing visualization of architectural designs in a standard format. Visualized architectures become more resilient to changes over time and increase the transparency of architectural processes.[35] In this way, it helps to overcome technical and organizational challenges in software projects.

Narwhal offers an integrated solution that makes software architecture and development processes more effective. The fact that diagrams can be easily shared and reused speeds up the workflows of software teams and increases efficiency. In addition, significant time savings are achieved in software development processes thanks to the automation of infrastructure configurations. This innovative approach provided by Narwhal allows for more successful management of both technical and

operational processes in software projects. Narwhal, which reduces the documentation burden by visualizing software architecture and mitigates the effects of architectural erosion, represents a significant paradigm shift in the world of software development.

4. INFRASTRUCTURE AS CODE AND ITS CHALLENGES

4.1 Infrastructure as Code : Revolutionizing Modern Infrastructure Management

In today's software development processes, the installation of the software on the server environment and the complete provision of the configurations required by this environment constitute one of the critical stages. In this context, automating the installation steps of the software on the target server not only helps to minimize manual errors, but also enables the processes to proceed more quickly and consistently. However, during the configuration of the target server, it is necessary to define various critical parameters such as creating a virtual private cloud (VPC), optimizing inbound and outbound port settings, and disk management. Such configurations play a fundamental role in creating the necessary infrastructure for the software to operate securely, scalably, and efficiently. The inclusion of Infrastructure as Code technologies in these processes has become an integral element of modern software development and DevOps practices.[36][37]

Today, the Infrastructure as Code (IaC) approach is widely adopted and used in modern software development and operations processes. This approach offers a paradigm that enables the definition and management of configurations of servers and related software tools through coding. Infrastructure as Code replaces traditional manual configuration processes, allowing the infrastructure to be created in a more consistent, repeatable and scalable way.[36][37] In particular, thanks to its ability to automate constantly repeated configuration processes, it aims to significantly reduce error rates and reduce maintenance costs in operational processes. In this context, Infrastructure as Code not only increases operational efficiency, but also contributes to the creation of a more reliable working environment by allowing software development teams to track and manage infrastructure changes through version control systems.

Today's popular software architectures, especially microservice-based approaches, create quite comprehensive and complex needs in terms of infrastructure and configuration. In microservice architectures, in line with the principle that each service can be developed and managed independently, these services require

infrastructure solutions tailored to their specific needs. For example, although each microservice has its own database and communication between services is provided through technologies such as message brokers, these structures make them quite dynamic and flexible, they also bring with them complex configuration requirements.

In this context, in an environment where different services may need different types of databases, the use of relational databases and NoSQL databases together may become inevitable. For example, while one microservice works with highly relational data, another may benefit from a NoSQL-based solution because it requires a flexible data model.[26] Similarly, the types of message brokers used in inter-service communication may vary depending on the characteristics of the services and the traffic density. Kafka is preferred in an application that requires high traffic and real-time data processing, while lighter solutions such as RabbitMQ may be suitable for lower traffic requirements.[38][39]

In addition, the assumption that each microservice can be written in different programming languages further increases the configuration requirements of the system. Services developed using different languages and frameworks will need to have customized configurations according to specific infrastructure needs. Considering all these factors, it is clear that infrastructure and configuration management in microservice architectures is an issue that needs to be addressed meticulously. While the management of these complex structures becomes possible with modern DevOps tools and automation techniques, the effective use of these tools provides a great advantage in terms of system performance and scalability.[40]

However, it is not enough to configure the software and tools properly, but also to distribute the services over multiple servers so that the architecture can effectively implement load balancing mechanisms. Such distribution is critical for both optimizing performance and increasing the scalability of the system. However, this approach also brings with it various infrastructural requirements and configuration needs.

In particular, effective use of load balancers in load balancing processes allows for the equal distribution of incoming requests among servers, thus preventing performance bottlenecks. In addition, the configuration of each server requires careful planning of network settings and connection protocols in order to ensure high

availability and reliability. While server configurations are addressed in a wide range from operating system settings to storage solutions, network configurations should be arranged to optimize the data transfer speed between servers.

In this context, it is clear that the infrastructure elements required for load balancing and scalability require a holistic approach that covers not only the management of hardware resources but also the communication processes between different components of the system.

This list will continue to grow. In this respect, since it is not possible to manage so many configurations manually today, development methods such as Infrastructure as Code have emerged and by making these configurations by writing code, processes have been automated to some extent.

The increasing complexity and diversity of today's software architectures reveal that, despite the valuable contributions provided by the Infrastructure as Code (IaC) approach to infrastructure management processes, this method may not provide a sufficient solution in every case. Especially in large-scale and dynamic software architectures, managing infrastructure configurations through code provides a systematic approach at the beginning, but creates significant difficulties over time as the complexity of the codes increases.

In this context, studies and observations show that IaC codes in large-scale software systems are becoming increasingly difficult to manage as they grow and need to adapt to changing needs. Especially in structures with many independent units such as microservice architectures, the existence of separate configuration requirements for each service and the constant updating of these structures are among the main factors that increase the complexity of IaC-based solutions. As a result, the process of defining infrastructure configurations with code increases automation and controllability at the beginning, but can become an element that makes maintenance and sustainability of codes difficult in the long run.[41]

Therefore, in modern software architectures, IaC alone may not provide a sufficient solution and the need to integrate additional approaches and tools for more complex needs comes to the fore. This situation brings with it the need for methods that take automation to higher levels in infrastructure management processes, provide

higher abstraction or include artificial intelligence-based recommendation and optimization systems.

4.2 Complexity Issues of Infrastructure as Code

Infrastructure as Code (IaC) has greatly accelerated software development processes thanks to the flexibility and automation it provides in the process of defining and managing infrastructure with code. However, this flexibility can also increase the complexity of infrastructures. Defining each component of an infrastructure through code can make these processes difficult to manage, especially when considering the scale and requirements of modern systems. For example, network topologies, firewall rules, database configurations, scaling policies, and identity management in an infrastructure all need to be addressed together. This requires writing hundreds or even thousands of lines of code and creating a structure that will maintain the consistency of this code.

Of course, the creation of this infrastructure code will also bring with it the cost of testing and debugging. Testing this process poses a great challenge. Since each component of the infrastructure is defined by code, comprehensive tests are required to ensure the integration and correct operation of the entire system. Especially in large and complex infrastructures, testing costs increase rapidly and detection of misconfigurations becomes more difficult. In fact, different tools and utilities are being developed just to meet the testing requirements.[41]

These challenges are especially evident when different layers of the infrastructure are interdependent. For example, in a microservices architecture, a separate database configuration may be required for each service, and specific network rules and security policies may be implemented for each service. This type of structure can affect the entire system when changes are made, and can increase the complexity of the code exponentially. Furthermore, when multiple teams work on the same codebase, errors and misunderstandings can occur due to lack of communication. This becomes more evident in environments where there is no harmony between teams, making infrastructure code management a serious problem in large-scale organizations.[42]

The complexity of IaC directly impacts not only the coding process but also the documentation of the infrastructure. Proper organization of each configuration file, clear definition of relationships, and keeping the files up-to-date are critical to the sustainability of the infrastructure. However, documentation processes can often be neglected or left incomplete due to pressure for rapid delivery or lack of resources. Incomplete documentation can make it difficult to understand the infrastructure over time, leading to misconfigurations and unexpected problems.[43][1]

Such deficiencies cause technical debt to accumulate and the infrastructure to become increasingly complex to manage. Technical debt is a situation where short-term solutions and temporary measures lead to long-term problems. For example, additions to the codebase as a temporary solution during a rapid update can increase the complexity of future changes. This causes the infrastructure to move away from the sustainable and scalable structure it was originally designed for, thus causing software architecture erosion at the infrastructure level.

At this point, while IaC aims to increase the sustainability and reproducibility of the infrastructure in theory, in practice it can have the opposite effect when mismanaged. In particular, rapid growth and disorganized management of the infrastructure code can accelerate this erosion.

As a result, to fully utilize the advantages of IaC, not only the writing of the infrastructure code but also its management, documentation and communication processes between teams must be effectively organized. Otherwise, the flexibility and automation advantages offered by IaC may be overshadowed by complexity and maintainability issues. At this point, visualization and schematization of the infrastructure can be an effective solution to prevent software architecture erosion.

4.3 Terraform

Terraform is an IaC tool used to define, manage, and monitor infrastructure resources.[44] Infrastructure can be defined and managed in a code-like language used by software development teams. Terraform uses its own proprietary configuration language, HashiCorp Configuration Language (HCL), rather than traditional configuration languages such as YAML or JSON.[44] HashiCorp Configuration

Language is a clear and understandable language that allows users to define infrastructure components and their relationships. This language allows for much faster tasks to be completed when writing code and allows for more consistent management of infrastructure definitions.

Terraform's configuration files are not only used to define infrastructure components; they can also be subject to version control. This allows infrastructure changes to be tracked and rolled back. In other words, a change in the infrastructure can be versioned, just like a code change in the software development process. In this way, every change made to the infrastructure becomes transparent and the history of each change can be tracked in detail. This feature enables debugging and healthier infrastructure management.

Terraform is generally used in cloud infrastructures, but it is not limited to cloud environments. It can be used to manage resources such as physical servers, network configurations, databases, etc. in both cloud environments (such as AWS, Google Cloud, Microsoft Azure, Oracle Cloud, DigitalOcean) and data centers.[44] This flexibility makes Terraform a tool that can manage infrastructure with the same configuration file on multiple platforms. In this way, if companies use multiple cloud providers, they can manage the infrastructure of these environments from a single code repository with Terraform.

Terraform uses configuration files to define the infrastructure. These files contain a set of resources, data sources, outputs, and variables. The most basic element of a Terraform configuration file is the resource block, which defines the infrastructure components.[44] These blocks define various resources, such as AWS EC2 instances, Azure virtual machines, load balancers, etc.

Terraform starts the plan phase to read and evaluate the defined configuration files. This phase checks if there are any differences between the current infrastructure and the definitions in the configuration file. If there are differences, Terraform lists these differences as a plan and presents them for user approval. This allows changes to be made to the infrastructure to be determined in advance. When the user approves, Terraform applies the changes and updates the infrastructure. Terraform stores each change to the infrastructure in a state file. This file represents the current infrastructure

state and allows Terraform to know what resources are available and when to change them.

This is the method that Terraform uses to declare and deploy infrastructure. Everything in the infrastructure is created and managed according to the rules specified in a Terraform configuration file. This approach provides much faster, more consistent, and more error-free management than traditional manual management.

4.3.1 Terraform and Its Contributions to Infrastructure as Code

Thanks to the Infrastructure as Code approach, software infrastructures and their configuration needs can be developed and managed using high-level programming languages.

Terraform is a widely used tool in the Infrastructure as Code field that aims to provide solutions to these challenges. Developed by HashiCorp, Terraform stands out for its ability to define infrastructure in code as well as implement these definitions in a platform-independent manner. It supports many providers such as AWS, Azure, Google Cloud, and manages the creation, modification, and deletion of infrastructure.[44]

In addition, Terraform has developed a library such as Cloud Development Kit for Terraform for many cloud service providers. These libraries can be used with 5 different programming languages.

These programming languages are listed below :

- C# .NET
- Python
- Typescript
- Java
- Go

Programming languages enable effective management of cloud infrastructure through Terraform's Cloud Development Kit for Terraform (CDKTF) library. CDKTF allows users to write infrastructure code (Infrastructure as Code) using programming

languages, which allows standardization of processes such as creating servers and performing necessary network configurations. This approach reduces manual intervention in infrastructure design, providing a more scalable and sustainable solution. An important advantage of CDKTF is that it facilitates the rapid and error-free deployment of complex infrastructures.

In addition, CDKTF is not limited to server configurations, but also has the ability to control container management tools such as Docker and Kubernetes. This feature allows developers to manage both container orchestration processes and infrastructure configurations from a single platform. Thanks to this integration, the flexibility and efficiency required by modern software development processes are provided, and the harmony between infrastructure and application management is increased. This versatile structure of CDKTF sets new standards in the code-based management of cloud infrastructures and increases efficiency in software engineering processes.

The main cloud service providers supported by Terraform are :

- Azure
- AWS
- Google Cloud Platform
- Oracle Cloud Infrastructure
- Alibaba Cloud

Cloud Development Kit for Terraform (CDKTF) adopts an open source development approach, incorporating not only Terraform's official service providers but also services provided by many open source communities. This open source ecosystem increases CDKTF's flexibility and extensibility, allowing users to meet a wide range of infrastructure needs. This provides a rich set of tools for creating and managing infrastructure code, using both official service providers and community contributions.

The development of this library is not limited to supporting existing cloud services, but also becoming a broad structure that increasingly encompasses more cloud services and DevOps tools. This growth allows CDKTF to play a critical role in modern software development and infrastructure management processes. This ever-expanding range of services facilitates users to integrate between different cloud platforms and

DevOps tools, while also providing the opportunity to quickly adapt to new technologies.

Thus, the growth of CDKTF based on an open source development model represents a significant transformation in the field of software and infrastructure engineering. Thanks to its expanding ecosystem, CDKTF has the potential to become not only a tool but also a standard for modern software development processes. This provides a more flexible and comprehensive infrastructure management experience for both individual users and corporate teams.

When configuring tools in a cloud system or a server, this process can be performed using a programming language using the Terraform CDKTF library, or this software infrastructure can be designed by writing and running Terraform configurations directly. Terraform provides a very important feature for both options, this feature is "state". When a software infrastructure is designed and run with Terraform, the relevant process is applied locally or in the server environment in accordance with the design, but if an addition is made to this configuration or an edit is made to a certain section and the code is run again later, Terraform only edits the existing structure for the relevant updates, eliminating the need to re-install the entire system. In this respect, Terraform makes a great contribution to the dynamic manageability of DevOps processes in the Infrastructure as Code approach.

Terraform provides two primary control mechanisms to ensure the correctness and consistency of changes made to infrastructure management: the "plan" and "apply" commands. These commands allow users to evaluate the effects of the Terraform configurations they have written in advance. Before a Terraform code is run, the "plan" command analyzes in detail which infrastructure components will be changed, which components will be added or removed, and reports them to the user. This process provides a critical pre-check mechanism in infrastructure management processes by allowing the user to verify that the changes are consistent with the desired results.

After a "plan" output is reviewed and the changes made are confirmed, the "apply" command is used to implement the necessary adjustments to the infrastructure components. This command allows Terraform to perform the installation and configuration operations specified for the relevant software infrastructure. By running the "apply" command, the user ensures that the previously planned changes are applied

directly, thus updating the infrastructure in accordance with the desired design. This process contributes greatly to minimizing errors during infrastructure management and to implementing changes in a controlled manner.

Although the use of the “plan” command is not mandatory, the activation of this control mechanism is especially important in terms of security and maintaining the stability of the infrastructure. Although it is possible to run the “apply” command directly, this method can be risky and lead to unexpected problems. Especially in large-scale systems or complex infrastructure configurations, such control mechanisms play a critical role in both reducing the error rate and maintaining the integrity of the software architecture. In this context, the use of the “plan” and “apply” commands together is an indispensable method for establishing best practice standards in infrastructure management.

Terraform’s “plan” and “apply” mechanisms allow for changes to be predicted. However, these advantages of Terraform can become a problem for complex infrastructures when not managed properly. Configurations with different modules and multiple dependencies can make code difficult to read and cause misunderstandings between teams. This shows the limitations of even a powerful tool like Terraform in reducing infrastructure complexity.[42]

As a result, despite the great conveniences it provides in infrastructure management, Terraform requires correct configuration and management. Especially in large and complex projects, modularization of the infrastructure and management of dependencies are critical. Although the module structures offered by Terraform offer the opportunity to make the infrastructure more modular, the dependencies and interactions between modules must be managed carefully. Otherwise, changes made in one module may have unexpected effects on other modules. In addition, configurations containing many sources and modules can make it difficult to ensure transparency and understandability of configuration files. This can make it difficult to track and debug changes in the infrastructure, thus affecting the collaboration of large teams and the sustainability of the infrastructure. At this point, although the "plan" and "apply" stages offered by Terraform play a critical role, additional measures and strategies must be developed for complex infrastructure management.

The Narwhal visual programming language adds a visual abstraction layer to Terraform's powerful infrastructure management capabilities built on top of the Hashicorp Configuration Language (HCL). Narwhal allows users to easily model infrastructure designs in a visual environment, while compiling these visual models to HCL, making the flexibility and power of the Terraform ecosystem directly accessible. This approach reduces the reliance on traditional text-based configuration files, providing a more understandable and accessible solution, especially for users new to infrastructure management.

Narwhal's visual abstraction capability not only makes it easier to write infrastructure codes, but also provides better comprehension through a visual representation of complex systems. Users can quickly modify designed infrastructure models and see the results of these changes directly in HCL format. Thus, Narwhal provides a user-friendly platform for both experienced infrastructure engineers and those new to the field, increasing productivity and accuracy in infrastructure management processes.

4.4 Reducing Complexity in Infrastructure as Code with Visualization

Visual programming tools play an important role in increasing the understandability and ease of interpretation of tasks to be performed or performed in many disciplines and application areas. These tools make complex algorithms or processes more accessible by being expressed with graphical representations. For example, the game development sector is a dynamic work area that brings together both technical and creative elements. In this sector, it is seen that visual programming tools are used effectively in addition to high-performance programming languages such as C++ and C#. These tools allow developers to structure algorithms and logical flows through visual components, making the development process both more intuitive and more efficient.

If we talk about C++, Bjarne Stroustrup, the developer of C++, stated in an interview that he could not remember all the features of C++ and could not answer some C++ questions without looking at the documentation or sources. [45] Game development with C++, which has a very solid place in the game development sector,

and the additional developments that the game will receive during its software lifecycle after development, can become quite complex. Or similar development processes may need to be repeated in other games to be developed.

In this context, the Blueprints visual programming language in Unreal Engine is of critical importance in game development processes. Blueprints provides a more intuitive and accessible development environment compared to traditional programming languages by offering the possibility of game development from a diagram using the drag-and-drop method. Each node creates an abstraction layer that can potentially represent tens or hundreds of lines of C++ code.[46][47] This approach offers game developers the opportunity to develop games more quickly and efficiently without delving into low-level programming details.

In addition, the visual diagram format of games developed with Blueprints greatly facilitates collaboration within the team and the adaptation process of new members. A new developer joining the team can quickly grasp the logical functioning of the code through visual diagrams and integrate into the project more quickly. The basis of the success of this structure is Blueprints' ability to automatically translate the definitions created on the diagram into C++ in the background. This translation process combines the convenience of abstraction with the power of a low-level language without any loss of performance, offering both ease of use and efficiency. For this reason, Blueprints stands out as an innovative tool that positively affects both functionality and team dynamics in game development processes.

A similar approach is now being used in the Unity platform, where the C# language is heavily used in game development processes, years after Blueprints was developed in Unreal Engine. Unity has developed a visual programming language called Unity Visual Scripting, which offers similar functionality to Blueprints. Similar to the structure where Blueprints performs abstraction by compiling diagrams into C++ language, Unity Visual Scripting contributes to the game development process by compiling the created visual diagrams into C# code.[48]

It is a generally accepted fact that C# is a more understandable and user-friendly programming language compared to C++. However, the development of a visual programming language for C#, which is relatively easier to learn and implement even on the Unity platform, shows that visual abstractions are becoming an increasing need

in the game development industry. This reflects a trend aimed at increasing not only the level of technical expertise but also the productivity and collaboration potential of developers.

Unity Visual Scripting has become a tool that appeals to both experienced developers and users with limited technical knowledge by enabling the definition of game mechanics and logical flows through a visual user interface. This type of abstraction not only speeds up the adaptation process of development teams to the project, but also allows for faster detection and resolution of errors. The development of Unity Visual Scripting has once again demonstrated the importance of visual programming tools in game development processes and increased the prevalence of such tools in the industry.

Outside of the gaming industry, the DevOps field, which is the combination of software development and operations processes, has undergone a significant transformation in recent years, making it necessary to use many new technologies and approaches. In this context, Docker containerization technology in particular has been widely adopted in modern software development and deployment processes and has become an indispensable part of DevOps. Docker has brought together all kinds of requirements for applications to run independently and in isolation, increasing the portability and scalability of the software, but at the same time creating complex challenges in configuring the system.

The basic principle of Docker is to provide isolated access to the operating system resources that applications need by running applications in a container. Each container operates in an environment that contains only its own requirements and dependencies, and does not interact directly with the host operating system. However, this isolation creates various configuration requirements for applications to communicate with each other and work efficiently. In particular, in order for multiple containers to work together, the port settings between the operating system and the container must be configured properly. This is a critical step for containers to communicate with each other and to be able to send and receive data from the outside world.

Another important configuration requirement is related to the allocation of disk space. Docker provides containers with the ability to allocate disk space independently of the operating system they are running on. These disk spaces are required for storing

data or managing configuration files for applications. However, improper setup of such disks can lead to data loss and degraded application performance.

Load distribution and more complex network requirements are another area of difficulty when using Docker. Especially for applications with high traffic requirements, multiple Docker containers need to be managed properly and load distributed effectively between them. This is where management tools like Docker Swarm come into play. Docker Swarm enables containers to be distributed across multiple servers and load balanced across these servers, ensuring high availability. However, when setting up Docker Swarm or similar network structures, all network configurations must be done correctly, and firewalls and port settings must be determined meticulously.

Since all these configurations need to be managed carefully, the installation and configuration process of Docker containers may seem simple at first, but can become quite complex over time. Careful planning of the interactions between system resources, port settings, disk allocations, and network configurations required by each container can bring potential risks that can negatively affect the performance and security of the system if not configured correctly. In this regard, a recent study addressed the automatic creation of Docker Compose configurations by designing them with a visual tool.

That is, visualization tools and visual programming languages have been used in many different software areas, and new tools and visual programming languages continue to be developed in these areas.

In this study, we developed the Narwhal visual programming language, just like Blue Print and Unity Visual Scripting and others, but targeting Infrastructure as Code, which directly concerns the design processes of DevOps and software architecture.

Narwhal is an innovative tool designed to bring a new dimension to the visual programming paradigm in modern software development processes. Developed within the scope of this work, Narwhal is based on the capabilities offered by visual programming tools such as BluePrint, Unity Visual Scripting, and more, but goes beyond them, focusing on the Infrastructure as Code (IaC) concept, which directly affects DevOps and software architecture processes. This focus aims to provide a more

intuitive and accessible solution that combines infrastructure management and software development processes.

Narwhal offers its users the ability to visualize infrastructure components through a diagram design screen. This screen allows visual modeling of the design of software tools (e.g. database, message queues, server configurations). Users can easily arrange infrastructure components using drag-and-drop, visualize the relationships between components, and directly convert these designs into code or configuration files. One of the most striking features of Narwhal is that it not only performs this conversion, but also automatically deploys and runs these configurations on targeted server environments or cloud services.

Narwhal's approach offers a significant innovation compared to traditional Infrastructure as Code applications. The often text-based and complex nature of IaC causes infrastructure managers and software developers to face difficulties in both the design and implementation phases. To overcome these difficulties, Narwhal abstracts the infrastructure components and makes them more accessible and understandable. Visualization not only provides a better understanding of the infrastructure, but also becomes a tool that strengthens communication and collaboration between teams.

Thanks to its visual programming approach, the abstraction layer provided by Narwhal enables software architects and DevOps teams to work on infrastructure design more quickly and flexibly. While users visualize the infrastructure design on diagrams, these designs are directly translated into executable Infrastructure as Code codes or configurations. Thus, the risk of errors that may occur during manual coding is minimized. In addition, Narwhal's automatic installation capabilities eliminate manual deployment processes, saving time and supporting continuous integration and continuous deployment (CI/CD) processes.

Another important contribution of Narwhal is that it appeals to a wider range of users by reducing the learning curve. While using traditional Infrastructure as Code tools usually requires certain technical knowledge and experience, Narwhal's visual interface makes these processes more intuitive to understand and use. This feature offers a great advantage not only for experienced software engineers and DevOps experts, but also for those who are new to infrastructure management.

As a result, Narwhal combines visual programming with Infrastructure as Code processes, offering significant innovations from both technical and operational perspectives. This approach, which abstracts infrastructure design and management by visualizing it, not only increases efficiency in modern software development processes, but also reduces the complexity of the infrastructure, enabling a wider range of users to participate in these processes. This study shows that Narwhal is not only a tool, but also has the potential to create a new paradigm in infrastructure management.

In order to make the possibilities provided by Infrastructure as Code more understandable and sustainable, visualization of the infrastructure is an important need. In this context, visual programming approaches such as Narwhal allow the infrastructure to be defined via a diagram and this diagram to be compiled directly into Terraform Hashicorp Configuration Language. Tools such as Narwhal enable even team members with less technical knowledge to contribute by diagramming the components and relationships of the infrastructure. In addition, such visualization prevents software architecture erosion and facilitates the long-term sustainability of the infrastructure.

Similar studies in this area have focused on visual diagram-based methods to enable more accessible and user-friendly design of Docker Compose configurations.[5] Developed to reduce the complexity of traditional text-based configuration processes, these approaches aim to make infrastructure design and management processes more intuitive. Visual tools allow components and relationships to be visualized on a diagram, making it easier to understand the general structure of the infrastructure and helping to prevent potential errors in the design process.[5] This provides a significant advantage, especially in complex systems where multiple components are integrated.

In these studies, it was determined that the time spent on the design process of configuration files using visual schemas was significantly reduced. In particular, the ability of users to create configurations through a visual interface prevented typographical and logical errors that could occur during manual editing. With this approach, both individual developers and teams were able to complete complex infrastructure designs in a shorter time, and the process became more efficient. In addition, the intuitiveness provided by visualization allowed users to better understand infrastructure designs and make more effective decisions on these designs.

Research shows that visual-based configuration tools not only save time, but also significantly reduce error rates. While the high error rates encountered in traditional methods can cause serious problems, especially in large-scale projects, such problems have been largely eliminated with visualization. For example, visually expressing the relationships between system components clearly allows for early detection of missing or faulty connections. This is an important factor that increases reliability in application and infrastructure processes.

As a result, the use of visual schema-based configuration tools provides significant benefits in infrastructure management and design processes. Time savings, reduced error rates, and more efficient use of development effort are among the main advantages of these approaches. It is also stated that visual methods facilitate communication and collaboration between teams. Studies in this area emphasize the importance of visualization in infrastructure management and provide a basis for larger-scale applications.

In addition, a similar visual modeling was compared with the Terraform equivalent Infrastructure as Code tool such as Ansible, and it was seen that the developed visual tools were easier to use and accelerated the development processes.[49]

In conclusion, Infrastructure as Code (IaC) management offers a revolutionary approach to modern software development and operations processes. This method provides a much faster, more efficient and error-free approach compared to manual infrastructure management methods by integrating the infrastructure into the software development processes. IaC provides a more reliable and repeatable infrastructure by writing infrastructure configurations as code and managing this code with version control systems. However, the implementation of this approach also brings with it some complexities and management challenges. As the size and complexity of the infrastructure increases, the challenges that arise in infrastructure management require the reshaping of organizational processes and collaboration methods.

Tools like Terraform offer powerful solutions to manage these complexities. These tools allow the infrastructure to be defined and managed as code, enabling rapid deployment and update of the system. However, for infrastructure management to reach its full potential, writing code alone is not enough. Visualizing and making the infrastructure understandable allows for more effective management of these

processes. Visualizing infrastructure configurations helps teams understand the overall structure of the system and the impacts of potential changes more quickly. Such visualizations make infrastructure management more transparent and accessible, resulting in fewer errors and faster returns.

At this point, innovative approaches such as Narwhal offer solutions that make infrastructure management more understandable and accessible. Narwhal strengthens collaboration between software developers and operations teams by providing visualization of the infrastructure. Visualized infrastructure helps analyze the effects of changes more quickly and accurately, which accelerates decision-making processes. In addition, such visualization methods allow for easier tracking of changes made to the infrastructure and create transparency in infrastructure management. Thus, improvements can be achieved in both technical and organizational processes.

As a result, the Infrastructure as Code (IaC) approach has become an integral part of modern software development processes. However, in order to fully benefit from the advantages offered by this approach, it is necessary to visualize the infrastructure and increase its understandability. In this context, innovative solutions such as Narwhal offer powerful tools that allow infrastructure management to improve not only from a technical perspective but also from an organizational perspective. Such solutions have a significant impact on transforming software development processes by making infrastructure management more efficient, sustainable and transparent.

4.5 Visual Programming Languages to Reduce Complexity in IaC

Today, we see the use of visual programming languages or visual tools in many areas. Visual programming languages are preferred for reasons such as reducing complexity in the relevant field, increasing development speed, etc.[47][50]

If we give examples of these areas;

- Game development (Blueprint, Unity Visual Scripting)
- Internet of Things and Robotics (VIPLE)
- Docker, Docker Compose [5]
- Data warehouse, database (Apache Nifi)

Visual programming languages or visual tools are tools that aim to make the software development process more accessible and understandable. These languages allow users to develop software through visual elements without having to deal with the complex syntax and logic structures of traditional text-based programming languages.[51]

In addition, visual programming is important not only for beginners or non-technical users in the software development process, but also for experienced and expert individuals in the field.[51] This approach makes the software development process more intuitive and accessible.

Such languages usually work with drag-and-drop logic, making it easier to understand and manage complex code structures.[51] In visual programming, instead of writing code, users add blocks or nodes that represent specific operations and connect them together. Each block represents an operation or set of instructions, and the connections between these symbols express the logic of the application. These visual elements help programmers develop software more quickly and accurately, while reducing the complexity of the software.

The ability of visual programming to reduce complexity is particularly evident in modeling the functional requirements of software. In traditional text-based programming languages, it is difficult to construct the logic of a program, use correct syntax, debug, and manage the code. However, in visual programming languages, users interact directly and visualize the logic of the application. This visualization makes complex algorithms and data flows more understandable and makes the operation of the code easier to follow.[50]

Another important advantage of visual programming languages is that they reduce the need for documentation. In traditional text-based languages, extensive documentation and explanations are required to understand the operation and logic of the software. However, in visual programming, the logic of the software is often directly represented through visual blocks, diagrams, or schematics. These visualizations clearly show how the code works and how it interacts with each other. Thus, no additional documentation is needed to understand the code. Users can quickly grasp the operation of the software through visual elements.[52]

As a result, visual programming languages make software development processes more accessible and understandable by abstracting the complexity of traditional text-based programming with visual elements.[51]

5. SOURCE TO SOURCE COMPILERS

A compiler is a software tool that converts source code into machine language. This software analyzes the syntax of a particular programming language and translates the written program into a format that can be executed by the machine. Compilers help in the error-free translation of source code into machine language by ensuring that programming languages are interpreted correctly. This process is the basic step that allows the program to run on the computer.

The function of a compiler is to process the source code it receives as input and produce output in a specific target language. The compiler creates a file that conforms to the requirements of the target language by recognizing all the symbols and constructs necessary for the software to function properly. This process involves not only correctly analyzing the syntax, but also correctly interpreting the semantics of the language. The compiler produces the output file by appropriately processing each statement in the source code.[53]

This process is done through a process called compilation. Compilation is the processing of input data received from the software within certain rules and the conversion into the appropriate output format. In this process, the compiler analyzes the source code line by line or block by block, identifies errors and converts the output into the target language format. Thus, the compilation process is a critical step in making the software workable.

Compilers are often described as software that converts source code into machine language, but this is not always the case. In the design of some programming languages, compilers do not directly convert to machine language. Instead, the source code is converted into a temporary intermediate language or layer. This approach allows for greater flexibility and optimization of the program execution process. In particular, a technique called "Just in Time" (JIT) compilation allows compilers to defer compilation until runtime and compile the code before it is run.

As an example, C# .NET and most other .NET-based languages allow the compiler to first convert source code into an intermediate layer called Microsoft Intermediate Language (MSIL). MSIL is a platform-independent language that is then compiled to run on the .NET Framework. This approach allows the compiler to

produce only platform-independent code, and allows optimizations specific to the target platform to be made later. MSIL provides software engineers who develop applications in the .NET environment with a more flexible and portable system.

Similarly, a similar method is applied in the Java programming language. The Java compiler converts the source code not directly into machine language, but into an intermediate language format called Bytecode. Bytecode is a format designed to be run by the Java Virtual Machine (JVM), and Java programs can be run on any platform as long as a JVM is present. This design allows software to run seamlessly on different platforms, in line with Java's "write once, run anywhere" philosophy.

A similar approach is used in the Erlang programming language, but here the middleware is a format called BEAM (Bogdan's Erlang Abstract Machine). The Erlang compiler translates the source code into the BEAM format, and these files are run by the Erlang virtual machine (BEAM). BEAM is a structure that supports the highly parallel processing capabilities of the Erlang language and its suitability for distributed systems. This offers significant advantages, especially for systems that require real-time and high performance.

This type of middleware allows compilers to not only produce machine language that is appropriate for the target platform, but also provides greater flexibility and optimization. Just in Time (JIT) compilation allows the necessary code to be compiled and optimized as the application runs, making the application more efficient during runtime. This allows only the required portion of the software to be compiled and run for the target platform, thus improving overall performance.

Using an intermediate language layer is an important feature that provides compilers with more dynamic features. For example, intermediate languages such as bytecode, MSIL (Microsoft Intermediate Language) and BEAM offer the ability to perform various analyses and changes at runtime. These languages provide an environment that can process metadata outside of the code generated during the static compilation process. Such a structure allows compilers to dynamically manipulate the code during operation, make optimizations and change the code when necessary. These dynamic features create an environment that provides flexibility and extensibility, especially in software development processes.

Another advantage of such intermediate languages is that dynamic features such as reflection and metaprogramming become available. Reflection allows a program to examine and change itself while it is running. Metaprogramming allows programs to write other programs or their own code. These features make the software development process much more flexible and powerful. For example, a program can obtain type information at runtime, establish relationships between classes, or modify object structures. This allows the software to gain new functions at runtime or to add features that have not been determined before.

Dynamic compilation provides a mechanism that makes such features more efficient. Intermediate languages allow routines created during the compilation process to be stored and dynamically linked when necessary. This allows the compiler to load and manage previously unspecified pieces of code or operations at runtime. Such dynamic features allow software to be more modular and flexible, because all components of the software can be developed and combined independently.

As a result, intermediate language layers play a critical role, especially in the creation of dynamic programming languages. Such languages offer customizable structures that can respond to changing data and code at runtime. These features allow developers to develop software with fewer restrictions and enable the development of more complex, dynamic applications. Thus, the software becomes more flexible and can be easily adapted to different situations. This also makes it easier to maintain and extend the software, because each module and component can be made functional separately as needed.

Using a middleware allows the complexity of the existing language to be managed through abstraction. This abstraction reduces the complexity of the software development process, making previously complex and time-consuming operations simpler. The middleware allows developers to focus on high-level features of the language, allowing them to obtain functional results in a more understandable and concise form. This process helps to produce a more efficient and faster solution during software development.

This kind of abstraction can significantly simplify the programming process by increasing the understandability of the input given to the compiler. Thanks to the

middleware, the developer has to deal with fewer complex details and can develop solutions for complex systems more quickly. As a result, the software development process is not only shorter and more efficient, but also more accessible and understandable. This dynamic structure offers a significant advantage that facilitates software engineering, especially in large and complex projects.

In addition, this compilation method can also provide a performance advantage. The procedure converted to the intermediate layer is called and executed directly. In this way, the procedures converted to the intermediate layer can be analyzed and in places where similar procedures are called repeatedly in different flows, an additional compilation process can be avoided, allowing the compiler to take faster action.

In general, compilers consist of two main parts.

These parts can be specified as:

- Backend
- Frontend

The frontend plays a critical role in compiler design because it is at the beginning of the compiler's work process and performs the initial analysis of the programming language source code received for compilation. This phase includes all the basic functions required for the compiler to work correctly. These functions include lexical analysis, parsing (syntax analysis), and syntax checks. The frontend part of the compiler is an important part that interprets and brings together the building blocks of the programming language. The work done in this section ensures that the source code is converted into an understandable and workable format.

The first and most important function of the frontend is to determine the tokens of the language. Tokens are the smallest meaningful units of a programming language and usually include keywords, operators, literals, variable names, and other language structures. These tokens are analyzed according to the syntax of the language and are associated with more complex structures in the next stage of the compiler. Token analysis reveals the basic structure of the language and the parsing process begins with this structure. Correctly determining the tokens is critical to

obtaining correct compilation results. An incorrect token definition can lead to errors in the next stages.

Parsing is the process of bringing together tokens and structuring them in accordance with the syntax rules of the language. In this process, logical relationships are established between tokens, and a structure that is appropriate to the grammar of the language is obtained. Parsing is usually performed with two main techniques: top-down and bottom-up parsing. Top-down parsing starts with predetermined grammar rules and combines input tokens under these rules to produce a solution. Bottom-up parsing, on the other hand, works with non-terminal symbols and combines lower-level structures with higher-level structures to obtain results. These two techniques offer different advantages depending on the structure and design of the language, and each is preferred in certain scenarios.

One of the most important steps in the parsing process is syntax checks. This stage checks the conformity of the language rules and detects errors. If a token does not conform to the language syntax, the compiler issues error messages and provides the developer with information about what is wrong with the code. This process is a critical step for the compiler to work correctly. Especially in complex languages, early detection of syntax errors can speed up the software development process and allow errors to be corrected at an early stage.

The next step is to create the Abstract Syntax Tree (AST), which is the output of the parsing process. The AST is a structural representation of the program and shows how the source code is organized in an abstract manner. The AST is a tree structure organized according to the grammar of the language. This tree contains all the elements that comply with the grammar of the language and represents the logical structure of the source code. The AST is used to provide data for the subsequent stages of the compiler, especially the optimization and targeting stages. The creation of the AST is the final step in the frontend part of the compiler and forms the basis for the subsequent stages.

When creating an AST, each node represents an element of the language, and the relationships between these nodes reveal the logic of the language. The AST can be customized for each language and may differ according to the language's specific grammar. This structure provides great convenience, especially to language

designers, in creating new languages and extending existing languages. The abstraction provided by the AST provides the compiler with a suitable environment for analyzing and processing the program at higher levels. Using the AST, the compiler can make the necessary optimizations in later steps and move towards the target code.

Another important concept in the frontend is the lexical analysis process. Lexical analysis is the process of analyzing the source code in the first step, and in this step, a lexer (or tokenizer) is used to separate the code into meaningful pieces. In lexical analysis, each character of the program is converted into a token. This process separates the keywords, symbols, and other language structures defined according to the syntax rules of the program. The lexer determines these tokens in the correct order and in accordance with the rules of the language. Lexical analysis is the first stage of the compiler process and includes the basic steps necessary for the correct processing of the program.

Parsing and lexical analysis are two basic components of the frontend, but both processes complement each other. Parsing works on the tokens provided by lexical analysis, while lexical analysis provides the data structure needed for parsing. Working together, these two processes allow the compiler to analyze the grammatical structure of the program accurately and efficiently. In addition, the interaction between these two stages allows the compiler to perform accurate error detection and rapid correction of errors.

This abstraction provided by the frontend enables other parts of the compiler to work more efficiently. Operations such as tokens, parsing and AST generation form the basis for the creation of the target code to be used in the next stages, optimization and debugging processes. The correct operation of the frontend contributes to the correct and fast execution of the remaining stages of the compiler.

In this way, a more robust and efficient operation is provided from the beginning to the end of the compiler process.

As a result, the frontend is an important component that forms the basic building blocks of a compiler. Operations such as lexical analysis, parsing, syntax checks, and the creation of the Abstract Syntax Tree are critical for the correct and efficient operation of the compiler. Each stage allows the compiler to analyze the

source code in accordance with the rules of the language, detect errors, and provide a correct data flow to the next stages. These processes are fundamental and inseparable steps in compiler design and are of great importance in the software development process.

Backend, as a stage in the compiler design, processes the analysis data obtained by the frontend and is a critical component that performs the operations required for the next stages. In the frontend part, the source code of the programming language is transformed into an abstract structure, usually in the form of an Abstract Syntax Tree (AST), by lexical analysis and parsing operations. This AST represents the structural and logical elements of the language and plays an extremely important role for use in the backend stage. The purpose of the backend is to take this abstract structure and transform it into a form suitable for a target processor or an intermediate layer.

The backend process involves converting the AST into targeted processor procedures by processing each node in a specific order. This stage usually provides targeting for a specific processor or middleware. For example, the backend can take the AST, perform mathematical and logical operations on it, and convert the results into a format that the processor can understand. The backend performs this conversion by paying attention not only to the processor-specific language rules but also to the performance optimizations of the target platform.

In other words, the processing performed in the backend requires a detailed processing process for each node of the AST. These nodes usually represent the basic building blocks of the language (e.g. expressions, operators, control structures, etc.). The backend analyzes each of these nodes in turn and creates the relevant procedures or intermediate layers according to the order of operations. This process is usually based on the order of operations determined according to a certain grammar rule and certain optimizations.

The targeted intermediate layer plays a very important role in this process. The backend part usually produces output suitable for the targeted processor or operating environment using an intermediate language. For example, an intermediate layer such as bytecode is used in Java and MSIL in C#. The backend's job is to transform the abstract structure obtained from the AST into this intermediate layer

and to reduce the abstraction layer in between to a level suitable for the processor. This intermediate layer then allows the program to be dynamically loaded and executed at runtime.

During the conversion process, the AST needs to be converted to a lower-level format. This process requires the backend of the compiler to make important decisions about how to organize the abstract representation of the language in physical memory. The backend takes into account efficiency and performance goals in terms of the logical structure of the processor or middleware while converting this abstract structure into a more suitable form. Memory management, data access patterns, and processor optimizations play important roles in this phase.

The backend also defines the procedures that must be performed to compile, specific to the processor architecture. These procedures determine the steps for running the program. This process ensures that the backend of the compiler produces instructions appropriate for the processor. These instructions are usually translated into a machine language or intermediate language format that the processor understands and processes. The backend also takes into account the portability of the language when generating processor instructions.

The conversion process can sometimes be made more efficient by using optimized intermediate layers. The backend can use a number of techniques to perform these optimizations. These can include loop optimizations, memory usage improvements, and microoptimizations at the processor instruction level. These optimizations aim to make the program run faster and use more resources more efficiently. In the backend phase, the compiler implements various strategies to maximize the performance requirements of the target platform.

Accordingly, the backend transforms the abstract structure obtained in the frontend phase into the target processor or middleware, thus providing the transition to the final stage of the compiler process. The most important function of the backend is to transform the abstract structure of the language into a form that can run on the target platform and to make it as efficient as possible. This transformation process is critical for the compiler to successfully produce code in the target language. The analysis and transformation processes performed on each node are optimized to ensure that the compiler works correctly and quickly.

As a result, backend design, as an important component of the compiler process, performs operations that comply with the grammar of the language and ultimately produces executable outputs that are suitable for the target platform. This phase includes important steps to ensure that the program runs efficiently, focusing not only on the operation of the language but also on factors such as performance, portability and optimization.

In this respect, although compilers appear to take a text input and produce another text as output, they actually contain a very complex and multi-layered structure. This process is not just a simple text conversion process, but also includes the necessity of producing outputs that are compatible with the syntax, semantic structure of the language and the target platform.

Programming languages are tools developed with designs specific to different problem areas, and these languages may have the ability to provide more effective solutions to certain problems. Each programming language can be used more efficiently and effectively in certain areas with its own syntax, structural features, and functional capabilities. These features are the determining factors in the design of the language and can affect the success of the language in a certain area.

For example, FORTRAN, which was developed for mathematical calculations and numerical analysis, can process complex calculations in this field quickly and efficiently.

Some programming languages offer high performance and ease of use in certain usage scenarios. Languages such as Java, C# and Go are preferred as powerful tools, especially in business-oriented software development. These languages have object-oriented programming features and provide sustainability, maintainable code structures and efficient work in large-scale software projects. Each of them is among the languages widely used in industrial software development and provides high performance while accelerating application development processes.

Erlang stands out with its loosely coupled structure and fault tolerance in areas that require high reliability and continuous operation, such as telecommunications systems. The design of this language provides great advantages in parallel processing, distributed systems and error management. Thanks to these features, Erlang ensures that other components continue to operate without being

affected in the event of an error in the system. This feature offers a great advantage, especially in sectors where uninterrupted service is critical, such as telecommunications.

As a result, the basis for programming languages to be more successful in certain areas is that the design and functionality of that language are shaped in accordance with the specific requirements of that area. These languages are developed and optimized to provide solutions to different problem areas, which makes them effective and widely usable in various industries.

However, the fact that the compiler structure is not easily developed can increase the cost of developing a programming language, especially in terms of time and effort. For this reason, various tools and infrastructures have been designed to facilitate the compiler development process. These tools help compiler designers overcome the difficulties they encounter and provide a more efficient development process. In this context, compiler infrastructures such as LLVM greatly facilitate the work of compiler designers.

LLVM is an infrastructure that provides significant convenience in the compiler development process and is particularly notable for its use of LLVM Intermediate Representation (IR). LLVM IR works similarly to a portable assembly language, offering compiler designers flexibility in creating compiler outputs specific to the targeted platform. This feature significantly reduces the cost of compiler development, especially saving time and effort in the work to be done in the backend part of the compiler. As a result, infrastructures such as LLVM accelerate the compiler development process and enable more efficient and effective software development.[54][55]

Similarly, various tools have been developed to facilitate front-end development of compilers. These tools simplify the process of defining the syntax of a programming language and processing the text inputs with this definition into a meaningful structure. For example, ANTLR (ANother Tool for Language Recognition) allows you to define the syntax of your own programming language. This tool analyzes the text inputs according to the defined syntax rules and converts them into an abstract syntax tree (AST) structure.[56][57]

ANTLR also provides library support that makes it easier to navigate the abstract syntax tree. This feature greatly simplifies the front-end development process, because compiler designers can manage the operations to be performed on the AST more efficiently. ANTLR provides a powerful tool for language designers, allowing them to perform syntax analysis and AST manipulation processes more quickly and flexibly in the front-end of the compiler. In this way, the compiler development process becomes both more efficient and more sustainable.[56][57]

Designing a compiler is still not an easy task, even though various tools and infrastructures have been developed today. While these tools make some of the front-end operations of the compiler easier, the back-end part of the compiler is usually a more complex and challenging process. In particular, the back-end part of the compiler can present major challenges depending on factors such as the target platform's processor architecture and operating system. In the back-end design, issues such as low-level optimizations, error management, and resource management are the elements that must be carefully addressed to ensure that the compiler operates efficiently and reliably.

The design challenges of a compiler become even more complex when the design needs to run on multiple processor architectures or operating systems. If the compiler has a cross-platform goal, that is, it needs to support different processor architectures (such as x86, ARM) or different operating systems (such as Linux, Windows, macOS), the design process becomes even more complex. Achieving such a goal requires considering the unique features of each platform. Although tools and infrastructures alleviate the complexity of backend design, generating code specific to each platform, ensuring compatibility, and optimization still present significant challenges to compiler designers. As a result, compiler design, although supported by modern tools, requires a complex and careful engineering process, especially when cross-platform goals are involved.

This is where source-to-source compilers make things a little easier. Source-to-source compilers are tools that make compiler design a little easier and speed up the development process. These types of compilers are usually called transpiler, transcompiler, or source-to-source compilers. The main function of source-to-source compilers is to convert the source code of one programming language to another programming language. These types of compilers take the source code and convert it

to its equivalent in another language, ensuring that the code is executable or understandable in a different environment.[58]

The working principle of these compilers is to convert an existing programming language, usually using their own compiler frontend. Source-to-source compilers analyze the program in the source language and reorganize it in accordance with the structures of the target language by analyzing the syntax of the existing language. In this way, programmers can translate code written in an existing language into another language, but this translation is usually done using the compiler frontend. In other words, source-to-source compilers can produce functional equivalents in another language without having to write the backend of the target languages.[59][53]

One advantage of such compilers is that they do not have to write the backend. Source-to-source compilers often use the backend of another compiler. This significantly reduces the complexity of compiler design, because the backend uses an already existing and tested infrastructure. Thus, compiler designers only have to deal with deciphering the syntax of the source language and translating it into the structure of the target language. This is an important factor in speeding up compiler development and reducing costs.

As a result, source-to-source compilers greatly facilitate compiler design. Instead of writing the backend of the target language from scratch, compiler designers can focus only on the frontend, using the infrastructure of another language. This saves time and effort, especially on large-scale projects. Source-to-source compilers make the compiler development process more efficient, while also facilitating integration between different languages and making the language translation process more accessible.[53]

This is a very common approach. One of the programming languages that implements this approach is the Dart programming language. The Dart code you wrote for the Flutter backend is actually translated into javascript with a source-to-source compiler. This javascript code is then run to run the application you developed.

Here, Dart integrated its frontend with JavaScript using a source-to-source compiler and thus avoided a large development cost. As is known, Javascript can be

used to develop programs in mobile, desktop and web environments. If a source-to-source compilation model were not applied, it would be necessary to develop a compiler backend that would target all these platforms and their operating systems. Naturally, developing a compiler backend that would cover all these platforms would involve a high cost.

There are many programming languages, not just the Dart programming language, that use this structure.

These programming languages :

Source-to-source Compiler	Source Language	Target Language
Babel	ES6+	ES5
Dart	Dart	JavaScript
ClojureScript	Clojure	JavaScript
Vala	Vala	C
Nim	Nim	C, C++, Objective-C, JavaScript
Mrustc	Rust	C
Fable	F#	JavaScript
Fable Python	F#	Python
Cfront	C++	C
Cerberus X	Cerberus	Javascript, Java, C++, C#

Table 3. Programming Languages That Used Source-to-source Compilers

In addition to text-based programming languages, visual programming languages can also use source-to-source compiler design. Visual programming languages allow users to create programs through graphical interfaces, and compiler designs for such languages often require a different approach. The application of source-to-source compilers to visual programming languages provides a very convenient solution for representing structural features and visual elements of these languages in target languages.[60]

Visual programming languages generally allow for interactive methods such as drag-and-drop coding, and are more suitable for such compiler designs. Because the visual tools that these languages offer to users can be easily understood and used even by people with less technical knowledge. Therefore, source-to-source compilers can enable users to program in a more accessible way by converting visual elements into

text-based languages. This offers an important advantage that facilitates the integration of visual programming with text-based languages.

For example, Google's Blockly visual programming language allows users to design programs using blocks. Blockly converts schematically organized block-based diagrams into text-based programming languages using a source-to-source compiler. This conversion allows the visual structures offered by Blockly to be adapted to the syntax of a specific target language. Thus, a program designed on a visual platform can be translated to run in text-based languages.[60]

Blockly can produce popular programming languages such as Lua, Python, PHP, JavaScript and Dart in its output. Users can convert the projects they create with the visual programming interface into code that can be run on different platforms. Thanks to its source-to-source compiler structure, Blockly's ability to convert between these languages allows users to improve their programming skills and increases the flexibility of visual languages. In this way, it makes it easier to transition to both the world of visual programming and traditional text-based languages.

Apart from this, the Blue Print visual environment within Unreal Engine produces an output in C++ language using a source-to-source compiler for the designed diagrams, facilitating game development. Similarly, there are visual environments in the Unity environment that can be compiled to C# language.

There are many node or block-based examples in the literature, including visual programming languages that compile to Lua.

These visual programming languages take advantage of the power of an existing text-based programming language and integrate the backend capabilities of this target language with their own visual syntax. In this way, instead of building the compiler backend of the target language from scratch, they create a more efficient structure by using the existing backend. Visual languages avoid the cost of backend development while also providing an abstracted and more user-friendly way of presenting programming.

In this process, visual programming languages offer the user an easier and more understandable diagram or schema design opportunity. By representing complex text-based codes with visual elements, users make programming accessible, especially for people with limited technical knowledge. Abstracted functionality with a visual

interface makes it easier to learn and implement programming, while also accelerating the software development process.

In addition, these structures are capable of using all the functionality of the programming language they target. Visual programming languages produce output in a way that is compatible with all the features of the target language they produce, thus offering users the advantages of both visual and text-based languages. As a result, such languages combine both the conveniences provided by visual diagrams and the powerful backend capabilities of the target language, making programming more efficient and user-friendly.

In other words, source-to-source compilers, in other words transpilers or transcompilers, offer a compiler design process that is similar to the capabilities of existing languages but less costly for the development of both visual and text-based programming languages.

The Narwhal visual programming language that we developed also includes a source-to-source compiler. Narwhal visual programming language offers a node-based drag-and-drop interface. Diagrams designed using this interface are compiled to the C# .NET target language by considering the relationships of each node one by one after the topological ordering used to determine node requirements. However, the compilation process is not limited to this. The C# .NET code produced according to the relevant diagram is recursively compiled with structures similar to the Razor web compiler and produces an output that can use C# .NET and Terraform CDKTF libraries. Finally, the Terraform Configuration Language output is obtained as a result of this structure being compiled, converted to MSIL and run. This Terraform Configuration Language output is compiled and has an executable structure, and Narwhal performs another compilation process and compiles and runs this output. Thus, the target software architecture or network etc. designed in the diagram. The configuration structure can be installed on the targeted local machines or servers, or even in the targeted cloud environments, the server can be launched and the necessary configurations can be made as indicated in the diagram.

Narwhal here abstracts the Infrastructure as Code structure repeatedly using a complex source-to-source compilation process, making infrastructure management more accessible and understandable. This approach allows infrastructure code to be

managed through a visual diagram. The visual representation is not only easier to understand than written code, but can also serve as an architectural documentation. Thus, users can easily follow the status and structure of their infrastructure on a visual platform.

The basis of Narwhal's capabilities is its source-to-source compiler functionality. The compilation process provides the abstraction required for infrastructure management and allows the user to program with a visual interface. All rules, relationships, and dependencies required for infrastructure components to work together are managed by these compilation processes. In this way, it is possible to control all the dynamics of the system with visual tools without delving into the lowest level technical details in infrastructure management.

With its source-to-source compilation functionality, Narwhal not only abstracts infrastructure codes but also creates an efficient environment for their maintenance and development. Through visual platforms, users adopt a more intuitive approach to understanding and managing complex infrastructure structures using visual diagrams and schemas rather than written codes. As a result, Narwhal provides an extensible solution that makes infrastructure management more efficient.

Narwhal visual programming language owes all of its functionality to its ability to compile from source to source.

In other words, when these and similar structures are taken into consideration, source-to-source compilers provide great advantages and conveniences to newly developed text or visual-based programming languages. In this respect, they play an undeniable role in compiler design.

As a result, source-to-source compilers stand out with their ability to bridge different platforms by increasing the level of abstraction in software development processes. These compilers transform code from one programming language to another, allowing you to benefit from the ecosystem of existing technologies and quickly adapt to new platforms. Especially in platform-independent solutions and modern application development scenarios, source-to-source compilers provide a significant advantage in terms of speed and efficiency. This ability allows software teams to deliver more effective solutions in a shorter time, thus reducing software development costs and encouraging innovation.

More importantly, source-to-source compilers simplify software development processes by providing developers with ease and offer flexible solutions that appeal to a wide range of users. Instead of complex backend-focused processes, developers can focus on the functionality and user experience of the software. These compilers eliminate the need to rewrite software for different platforms, ensuring maintainability and scalability. In terms of software engineering, source-to-source compilers are not only a technical tool, but also a strategy that has the potential to revolutionize software development processes. Therefore, the role of these compilers will increase in the software ecosystem of the future.

6. VISUAL PROGRAMMING

6.1 Purpose and Vision of the Narwhal Visual Programming Language

Narwhal visual programming language was developed to eliminate or contribute to the elimination of devops-related causes of erosion problems in software architectures that we frequently encounter today by schematizing Devops processes and facilitating their understandability.

Narwhal is a node-based visual programming language. You can drag and drop the required nodes onto the design screen and design the necessary cloud infrastructure development by defining the connections between them.

Thanks to Narwhal's node-based design structure, the design created for the infrastructure will also be a visual and schematic version of the software infrastructure with all its details, thus reducing the need for documentation in software projects. Thus, it will reduce the effect of possible software architecture erosions that may arise from lack of documentation due to technical or non-technical reasons in software projects that have to undergo constant change due to the software life cycle over time, or prevent their occurrence.

In addition, since it will have an easily shareable and understandable structure thanks to its visual structure, it will reduce the impact of architectural erosions caused by lack of orientation resulting from the change of team members working on software projects or the increase in employee circulation within the company.

Thanks to this easily shareable structure, understanding the architecture will be fast and easy, so understanding the architecture in open source projects will become easier and the speed of development of applications from an architectural perspective will accelerate.

Most importantly, we offer an effective solution against the architectural erosion that will occur in software projects due to infrastructural decisions made by non-technical people. Because, as is known, the biggest reason for architectural erosion in software projects is that non-technical people make critical decisions for software architecture, rather than technical decisions.[2] Since creating the

infrastructure on a visual diagram will make it easier for non-technical people to understand the infrastructure, Narwhal offers an effective solution here.

6.2 Architecture of Narwhal Visual Programming Language

Narwhal basically consists of 3 applications.

These are :

- Narwhal API
- Narwhal Web User Interface
- Narwhal Desktop Application

The design you create on the web interface is transmitted to the Narhwal API via HTTP requests, and the API performs the necessary compilation on its side. Then, using the Desktop application, you can launch the structure you have previously designed in the interface environment on your own computer or on a remote server.

The web interface allows you to examine the architectural designs shared by others and, if necessary, take a copy and improve it, or easily share the infrastructure you have developed.

6.2.1 Web User Interface of Narwhal Visual Programming Language

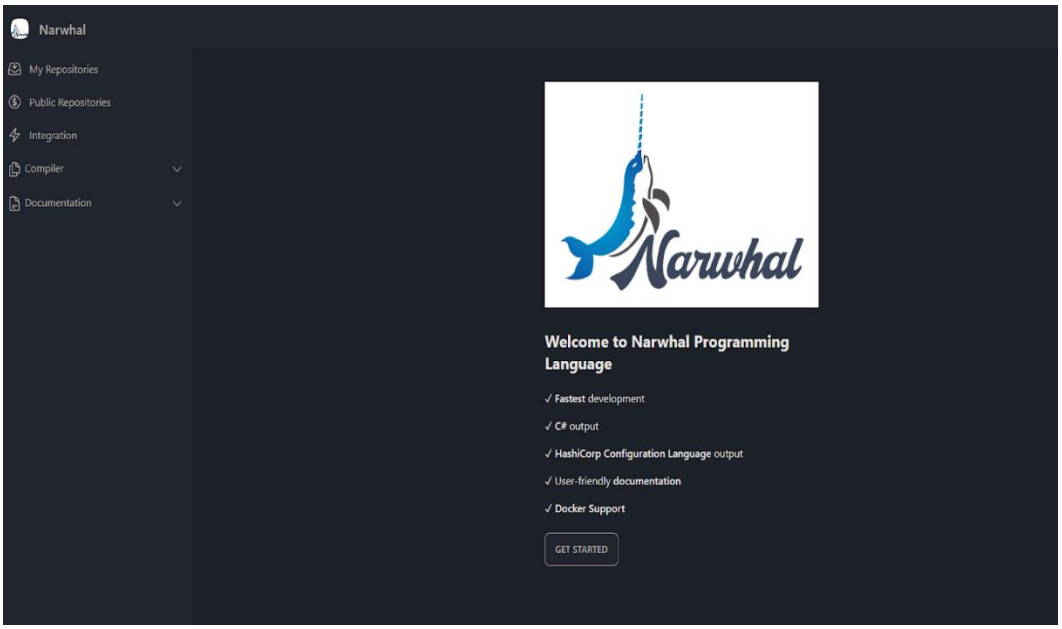

Figure 7. Narwhal Web Main Screen

As indicated in Figure 7, you can get the C# output or HashiCorp Configuration Language output of the design.

If we examine the design screen in the web interface:

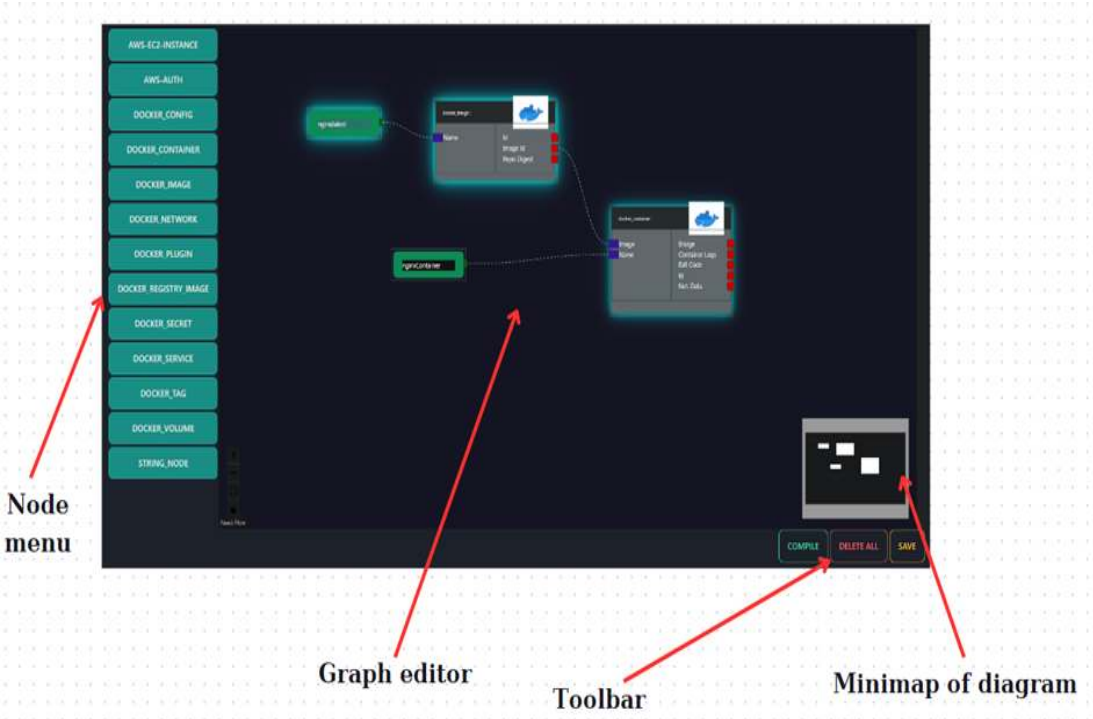

Figure 8. Narwhal Web Designer

Node menu The section on the left side of the screen that lists the nodes that can be used in the graph editor.

Graph Editor The area in the middle of the screen where different configurations can be made using the nodes in it and where the visual language interacts.

Toolbar The area on the bottom right of the screen that contains functions that will allow the work done in the Graph editor to be compiled, deleted or saved.

Minimap The map on the bottom right of the screen that shows the nodes used in the Graph editor in a wide perspective.

The node structure and connections designed on the Graph Editor are transmitted to the API by the interface and the compile process is performed on the API side.

In the design seen in Figure 8, the docker container is launched via an image specified for nginx.

6.2.2 Desktop Application of Narwhal

Figure 9. Narwhal Desktop Application Login Screen

Designed infrastructure and configurations may need to be run frequently on a local computer, and Narwhal offers a desktop application for this exact need. You can access the infrastructure diagrams you designed on the web using your login information through this application. A list of the diagrams you designed will be displayed on the screen when you successfully log in.

Figure 10. Narwhal Desktop Application Main Screen

As seen in Figure 10, the infrastructure configuration we designed in Figure 8 is listed on the main screen.

If we have done different studies in the web interface and want to view these studies in the desktop application, it will be enough to click the "Refresh" button seen in Figure 10. It will bring the current studies from the API to the main screen.

When the required work appears on the main screen of the desktop application, you can first select the project to be run in the list seen in Figure 10 and then click the "RUN" button to have the relevant configuration compiled and run directly on Docker.

You can also obtain a ready-to-run output of your design using Terraform CDKTF with C# .NET without compiling it on the desktop application. You can make changes to it. Thus, Narwhal adopts a more dynamic approach by ensuring that its own features can be easily bypassed and defines a wider area of action for the user.

You can see the C# .NET output of the project designed in Figure 8 and seen in Figure 10 below in Figure 11.

```
namespace MyCompany.MyApp {
    class MainStack : TerraformStack {
        public MainStack(Construct scope, string id) : base(scope, id) {
            new DockerProvider(this, "docker", new DockerProviderConfig { });
            string n_7191bc26328d45999672e10acfb22798 = "myNginxContainer";
            string n_e2acc9fa4efb49efb0750273c7b5b038 = "nginx:latest";
            var n_ecfa602eff094b699d6a3814326c9e09 = new Image(this,"Image1f6c026f-d6d4-46b8-85c3-ffd27f750bc6",
            new ImageConfig{ Name=n_e2acc9fa4efb49efb0750273c7b5b038,});
            var n_1c02ac250a24482ab28a4d42d1fa4f8f = new Container(this,"Container3689ac41-e404-4b33-acd8-67309e4af69f",
            new ContainerConfig{Image=n_e2acc9fa4efb49efb0750273c7b5b038,Name=n_7191bc26328d45999672e10acfb22798,});
        }
    }
}
```

Figure 11. Generated C# .NET code

We will go into detail about how this code is created when explaining Narwhal's source-to-source compiler structure. Here, the user can easily access and change the intermediate products or results from all interface screens.

6.2.3 Compiler Architecture of Narwhal Visual Programming Language

Narwhal uses a source-to-source compiler in its own structure. The diagram designed in the web interface is converted into a complex and detailed json format. This json is then transmitted to Narwhal's APIs and compiled there and converted to HCL (HashiCorp Configuration Language). In the final stage, this HCL is run to create the necessary infrastructure.

Let's examine these steps in detail. For the structure designed in Figure 8, a json containing the Nodes and Edges values will be generated in the interface.

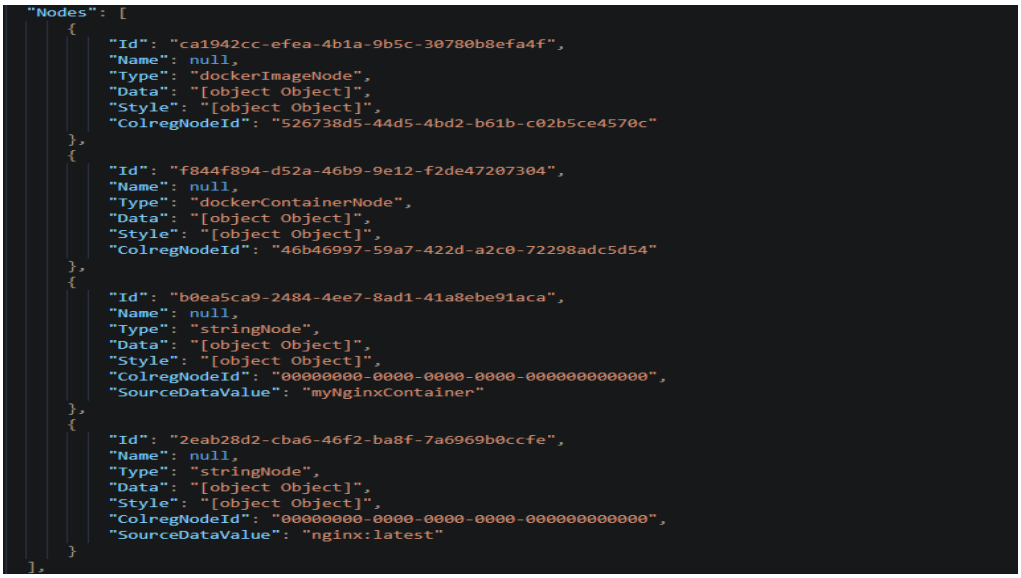

Figure 12. Nodes In Json Model That Generated by Narwhal Web UI

As seen in Figure 12, each node has an Id value, which acts as a unique identity for all elements in the interface.

The Type field serves as an identity that is recognized by the Narwhal compiler for the relevant node and allows the compiler to recognize and process the node's properties during the compile phase. By recognizing this Type field, Narwhal knows what properties the relevant node has.

The ColregNodeId field allows Narwhal to access the properties of the node, just like the Type field. However, this access is more detailed than the Type and is an identification information that defines all the technical properties for the relevant node. It is normal for nodes with the same Type value to have the same ColregNodeId values. However, all elements must have a unique Id field.

Of course, a complex diagram cannot be specified with just nodes. The relationship between these nodes is the element that forms the main substructure. There is also an Edges field in the JSON value, just like Nodes. This field defines the relationships between the nodes in the diagram.

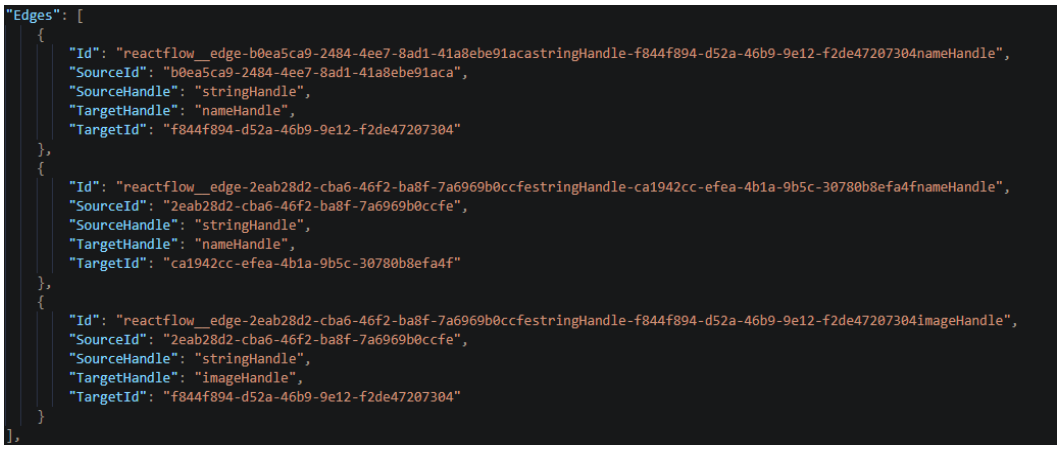

Figure 13. Edges In Json Model That Generated by Narwhal Web UI

As seen in Figure 13, each node has an ID value and each of the connections between nodes also has a unique ID value.

The **SourceId** field indicates the unique ID of the node from which the connection was initiated in the connection between nodes.

The **TargetId** field indicates the unique ID of the target node in the connection between nodes.

Of course, a node can have more than one connection point. In order to understand which of these connection points is connected to which, it is not enough to know the identity of the connected nodes. Therefore, the Source and Target Handle values are created. These are the information that indicates which connection point of the relevant node is connected to which port on the target node.

As soon as there is a change in the interface diagram, it models this design created within itself directly into a json as we explained. After the design is finished, when the user wants to compile it, he presses the “Compile” button indicated in Figure 5 and the relevant model is sent to Narwhal’s API via HTTP request. After this stage, the compilation process of the json model starts.

When the request is forwarded to Narwhal APIs, first the authorization check is performed. Then, the relevant model is deserialized and assigned to the models within Narwhal.

One of the biggest problems in processing this model is knowing which node and relationship to start the compilation process on. To determine this, Narwhal uses a topological ordering system.

Topological Sorting is the problem of sorting the nodes of a Directed Acyclic Graph (DAG). This sorting is performed in accordance with the graph's connections, i.e., dependency relations. In other words, if there is a directed connection from one node (e.g. A) to another (e.g. B), then node A should appear before node B in the sorting.

This method is especially useful when dependencies are critical. For example, it is ideal for determining the priority of a set of tasks, organizing the compilation order of a software, or modeling a data flow.

Narwhal implements this method as follows. If a node needs another node to be compiled (because it is connected), the node it needs is compiled first. In order to perform this operation, it performs a topological sorting using the nodes and their connections in the model coming from the interface.

Then, by opening a scope at the compiler level, it determines the nodes that are defined as nodes but actually hold primitive values interactively. These can contain numerical values defined for Port, string values defined for naming, or true/false boolean value types. These nodes are recognized by Narwhal by looking at the Type field that we mentioned while explaining the Json model and are kept in a memory area with their values and connections before the compilation process.

The compiler obtains a list of sorted nodes from the topological sorting result. Before processing this list, it stores the nodes containing the primitive value it has determined in the memory area and does not include these nodes in the topological sorting list because they are static elements that do not need to be sorted.

All nodes and their connections in the list are analyzed one by one with a loop. Nodes are recognized by their identity information and compiled into C# .NET code according to the principles of the connections they contain. During this compilation, some of the abstracted versions of the node's features on the cloud and devops side are compiled using the Razor Engine.

Finally, after this successful compilation process, the compiled code and the information of the relevant project are saved in the database. Thus, users can access

the compiled version of the diagram via Web or Desktop interfaces and run the infrastructure they designed.

After all these compilation processes, the user can obtain the diagram he designed, written in C# .NET or HCL (HashiCorp Configuration Language) output. Or, he can just create the infrastructure he designed without dealing with all these codes. The user does not need the C# and HCL code produced as a result of the compilation to create the infrastructure. These are only features provided to the user to enable them to take more dynamic actions by penetrating Narwhal's system. In this way, even if Narwhal is not used as the main product, it can be used as a tool to write the code necessary to create certain code structures and manage the infrastructure.

```
{
  "//": {
    "metadata": {
      "backend": "local",
      "stackName": "Area24",
      "version": "0.20.9"
    },
    "outputs": {
    }
  },
  "provider": {
    "docker": [
      {
      }
    ]
  },
  "resource": {
    "docker_container": {
      "Container3689ac41-e404-4b33-acd8-67309e4af69f": {
        "//": {
          "metadata": {
            "path": "Area24/Container3689ac41-e404-4b33-acd8-67309e4af69f",
            "uniqueId": "Container3689ac41-e404-4b33-acd8-67309e4af69f"
          }
        },
        "image": "nginx:latest",
        "name": "myNginxContainer"
      }
    },
    "docker_image": {
      "Image1f6c026f-d6d4-46b8-85c3-ffd27f750bc6": {
        "//": {
          "metadata": {
            "path": "Area24/Image1f6c026f-d6d4-46b8-85c3-ffd27f750bc6",
            "uniqueId": "Image1f6c026f-d6d4-46b8-85c3-ffd27f750bc6"
          }
        },
        "name": "nginx:latest"
      }
    }
  },
  "terraform": {
    "backend": {
      "local": {
        "path": "C:\\Users\\Eminf\\Area24\\terraform.Area24.tfstate"
      }
    },
    "required_providers": {
      "docker": {
        "source": "kreuzwerker/docker",
        "version": "3.0.2"
      }
    }
  }
}
```


Figure 14. Generated HCL Code After Compilation

With these features, Narwhal can directly launch the structure designed in Figure 8. Normally, an HCL configuration would be needed as indicated in Figure 14 or a C# .NET code as indicated in Figure 11. Narwhal abstracts all this library, programming language and terminal information, allowing the user to focus only on the features of the cloud system and the architecture they will design.

With its current version, Narwhal supports operations that can be performed on AWS and Docker.

7. CAPABILITIES OF NARWHAL

7.1 Example Use-Cases of Narwhal

Below are examples of Narwhal's usage. It can be utilized for deploying a monolithic or microservices-based application infrastructure, as well as for configuring the server hosting these applications.

7.1.1 Monolith Architecture Example

This example demonstrates how to design the infrastructure of a simple monolithic architecture software using Narwhal.

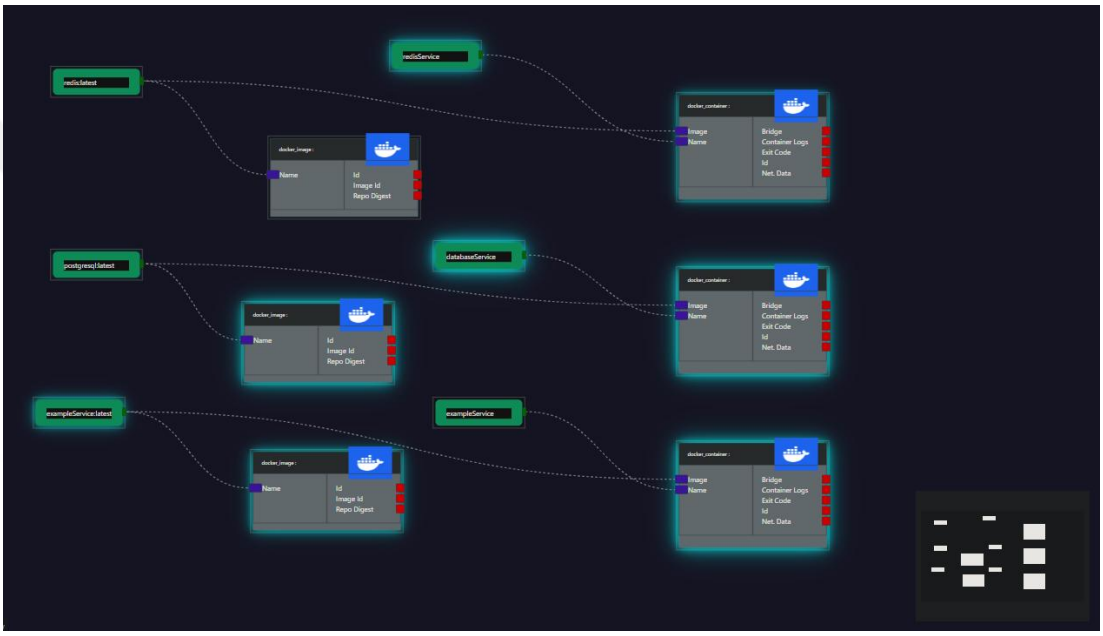

Figure 15. A Simple Monolith App Example With Narwhal

As shown in Figure 15, a Redis in-memory database has been deployed for caching relevant operations, while a PostgreSQL database has been set up for persistent data storage. Additionally, an example application, exampleService, which utilizes both databases, has been deployed.

The database and the exampleService application images are publicly accessible on Docker Hub. Narwhal pulls these images from Docker Hub, deploys them as Docker containers, and configures them according to the naming conventions specified in the diagram. By replicating the database and increasing the number of exampleService instances as depicted in the diagram, we can establish a representative microservices architecture.

7.1.2 Run Application From Private Registry

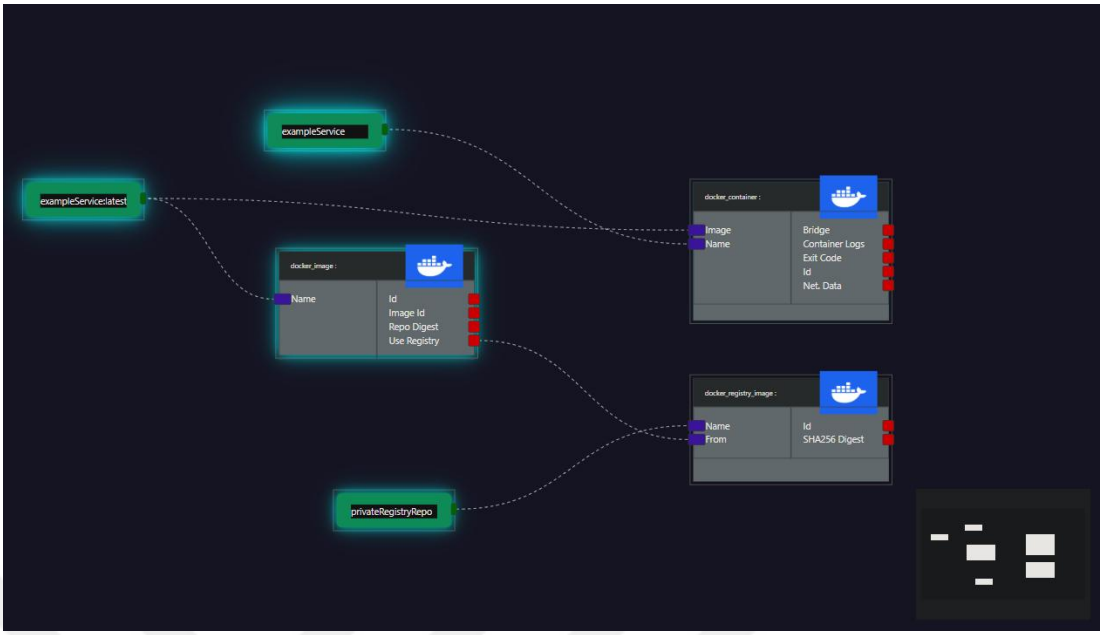

Figure 16. Private Docker Registry Example

Normally, application images are retrieved from a public registry when deploying them. However, in this example, Narwhal is used to pull the application image from a private registry repository and deploy a running Docker container from it. In cases where multiple private registries are needed, registry nodes can be replicated and utilized within a shared schema.

7.1.3 Running An Application On Amazon Web Services Example

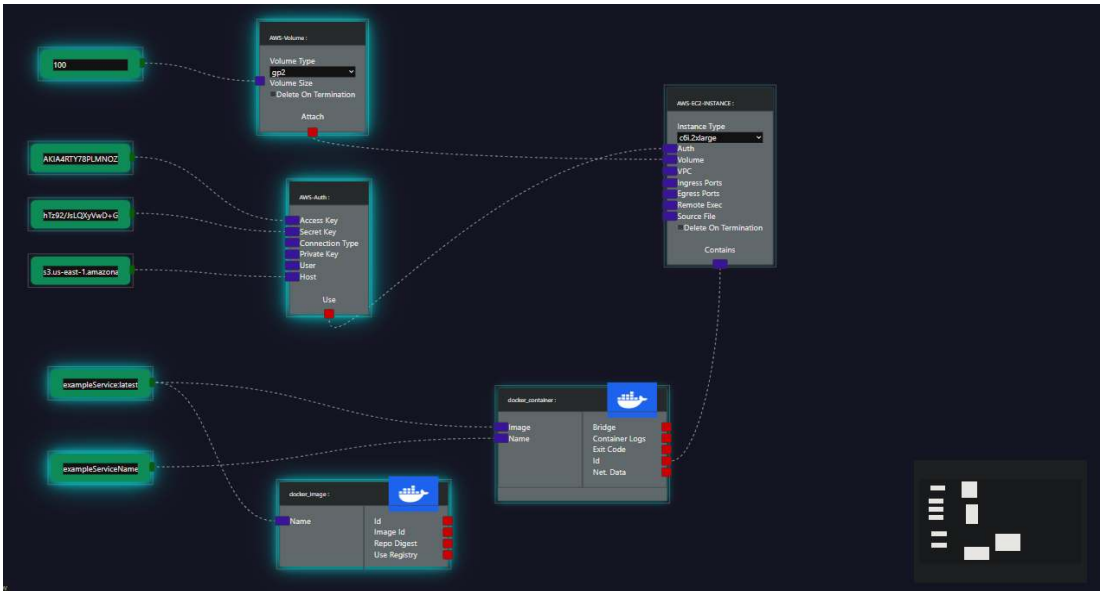

Figure 17. Running On Amazon Web Services

In this example, a Docker container was created from a Docker image. Subsequently, a c6i.2xlarge instance was launched in a cloud environment after logging in with AWS authentication credentials. A 100 GB gp2 disk was allocated to this instance. Finally, the previously created container was deployed on the designated Amazon Web Services server.

7.2 Supported Functions by Narwhal

When discussing the capabilities of Narwhal, it possesses comprehensive features at the Docker level, including Config, Container, Service, Network, Plugin, Secret, Tag, Volume, and Registry.

7.2.1 Docker Config

It has the ability to manage the configurations of docker services created from a docker image within the network they create with docker swarm. It can receive and process config data as base64.

data (String) Base64-url-safe-encoded config data

name (String) User-defined name of the config

7.2.2 Docker Container

By taking a docker image, you can start a running container from this image and optionally support the following docker container features.

attach (Boolean) If true attach to the container after its creation and waits the end of its execution. Defaults to false.

capabilities (Block Set, Max: 1) Add or drop certain linux capabilities.

cgroupns_mode (String) Cgroup namespace mode to use for the container. Possible values are: private, host.

command (List of String) The command to use to start the container. For example, to run /usr/bin/myprogram -f baz.conf set the command to be ["/usr/bin/myprogram","-f","baz.con"].

container_read_refresh_timeout_milliseconds (Number) The total number of milliseconds to wait for the container to reach status 'running'

cpu_set (String) A comma-separated list or hyphen-separated range of CPUs a container can use, e.g. 0-1.

cpu_shares (Number) CPU shares (relative weight) for the container.

destroy_grace_seconds (Number) If defined will attempt to stop the container before destroying. Container will be destroyed after n seconds or on successful stop.

devices (Block Set) Bind devices to the container.

dns (Set of String) DNS servers to use.

dns_opts (Set of String) DNS options used by the DNS provider(s), see resolv.conf documentation for valid list of options.

dns_search (Set of String) DNS search domains that are used when bare unqualified hostnames are used inside of the container.

domainname (String) Domain name of the container.

entrypoint (List of String) The command to use as the Entrypoint for the container. The Entrypoint allows you to configure a container to run as an executable. For example, to run /usr/bin/myprogram when starting a container, set the entrypoint to be "/usr/bin/myprogra"].

env (Set of String) Environment variables to set in the form of KEY=VALUE, e.g. DEBUG=0

gpus (String) GPU devices to add to the container. Currently, only the value all is supported. Passing any other value will result in unexpected behavior.

group_add (Set of String) Additional groups for the container user

healthcheck (Block List, Max: 1) A test to perform to check that the container is healthy

host (Block Set) Additional hosts to add to the container.

hostname (String) Hostname of the container.

init (Boolean) Configured whether an init process should be injected for this container. If unset this will default to the dockerd defaults.

ipc_mode (String) IPC sharing mode for the container. Possible values are: none, private, shareable, container:<name|id> or host.

labels (Block Set) User-defined key/value metadata

log_driver (String) The logging driver to use for the container.

log_opts (Map of String) Key/value pairs to use as options for the logging driver.

logs (Boolean) Save the container logs (attach must be enabled). Defaults to false.

max_retry_count (Number) The maximum amount of times to an attempt a restart when restart is set to 'on-failure'.

memory (Number) The memory limit for the container in MBs.

memory_swap (Number) The total memory limit (memory + swap) for the container in MBs. This setting may compute to -1 after terraform apply if the target host doesn't support memory swap, when that is the case docker will use a soft limitation.

mounts (Block Set) Specification for mounts to be added to containers created as part of the service.

must_run (Boolean) If true, then the Docker container will be kept running. If false, then as long as the container exists, Terraform assumes it is successful. Defaults to true.

network_mode (String) Network mode of the container.

networks_advanced (Block Set) The networks the container is attached to

pid_mode (String) he PID (Process) Namespace mode for the container. Either container:<name|id> or host.

ports (Block List) Publish a container's port(s) to the host.

privileged (Boolean) If true, the container runs in privileged mode.

publish_all_ports (Boolean) Publish all ports of the container.

read_only (Boolean) If true, the container will be started as readonly. Defaults to false.

remove_volumes (Boolean) If true, it will remove anonymous volumes associated with the container. Defaults to true.

restart (String) The restart policy for the container. Must be one of 'no', 'on-failure', 'always', 'unless-stopped'. Defaults to no.

rm (Boolean) If true, then the container will be automatically removed when it exits. Defaults to false.

runtime (String) Runtime to use for the container.

security_opts (Set of String) List of string values to customize labels for MLS systems.

shm_size (Number) Size of /dev/shm in MBs.

start (Boolean) If true, then the Docker container will be started after creation. If false, then the container is only created. Defaults to true.

stdin_open (Boolean) If true, keep STDIN open even if not attached (docker run -i). Defaults to false.

stop_signal (String) Signal to stop a container (default SIGTERM).

stop_timeout (Number) Timeout (in seconds) to stop a container.

storage_opts (Map of String) Key/value pairs for the storage driver options, e.g. size: 120G

sysctls (Map of String) A map of kernel parameters (sysctls) to set in the container.

tmpfs (Map of String) A map of container directories which should be replaced by tmpfs mounts, and their corresponding mount options.

tty (Boolean) If true, allocate a pseudo-tty (docker run -t). Defaults to false.

ulimit (Block Set) Ulimit options to add.

upload (Block Set) Specifies files to upload to the container before starting it. Only one of content or content_base64 can be set and at least one of them has to be set.

user (String) User used for run the first process. Format is user or user:group which user and group can be passed literraly or by name.

userns_mode (String) Sets the usernamespace mode for the container when usernamespace remapping option is enabled.

volumes (Block Set) Spec for mounting volumes in the container.

wait (Boolean) If true, then the Docker container is waited for being healthy state after creation. If false, then the container health state is not checked. Defaults to false.

wait_timeout (Number) The timeout in seconds to wait the container to be healthy after creation. Defaults to 60.

working_dir (String) The working directory for commands to run in.

7.2.3 Docker Image

Pulls a Docker image to a given Docker host from a Docker Registry. Optionally supports the following Docker container features.

build (Block Set, Max: 1) Configuration to build an image. Please see docker build command reference too.

force_remove (Boolean) If true, then the image is removed forcibly when the resource is destroyed.

keep_locally (Boolean) If true, then the Docker image won't be deleted on destroy operation. If this is false, it will delete the image from the docker local storage on destroy operation.

platform (String) The platform to use when pulling the image. Defaults to the platform of the current machine.

pull_triggers (Set of String) List of values which cause an image pull when changed. This is used to store the image digest from the registry when using the docker_registry_image.

triggers (Map of String) A map of arbitrary strings that, when changed, will force the docker_image resource to be replaced. This can be used to rebuild an image when contents of source code folders change.

7.2.4 Docker Network

It has the ability to create networks within Docker and manage the configurations of the relevant network. It can be customized as follows.

attachable (Boolean) Enable manual container attachment to the network.

check_duplicate (Boolean) Requests daemon to check for networks with same name.

driver (String) The driver of the Docker network. Possible values are bridge, host, overlay, macvlan. See network docs for more details.

ingress (Boolean) Create swarm routing-mesh network. Defaults to false.

internal (Boolean) Whether the network is internal.

ipam_config (Block Set) The IPAM configuration options

ipam_driver (String) Driver used by the custom IP scheme of the network. Defaults to default

ipam_options (Map of String) Provide explicit options to the IPAM driver. Valid options vary with ipam_driver and refer to that driver's documentation for more details.

ipv6 (Boolean) Enable IPv6 networking. Defaults to false.

labels (Block Set) User-defined key/value metadata

options (Map of String) Only available with bridge networks. See bridge options docs for more details.

7.2.5 Docker Plugin

It has the ability to add and use features that Docker Engine does not have, via plugins. It optionally supports the following configurations.

alias (String) Docker Plugin alias

enable_timeout (Number) HTTP client timeout to enable the plugin

enabled (Boolean) If true the plugin is enabled. Defaults to true

env (Set of String) The environment variables in the form of KEY=VALUE, e.g. DEBUG=0

force_destroy (Boolean) If true, then the plugin is destroyed forcibly

force_disable (Boolean) If true, then the plugin is disabled forcibly

grant_all_permissions (Boolean) If true, grant all permissions necessary to run the plugin

grant_permissions (Block Set) Grant specific permissions only

7.2.6 Docker Registry Image

Manages the lifecycle of docker image in a registry. You can upload images to a registry and also delete them again. Optionally supports the following configurations.

insecure_skip_verify (Boolean) If true, the verification of TLS certificates of the server/registry is disabled. Defaults to false

keep_remotely (Boolean) If true, then the Docker image won't be deleted on destroy operation. If this is false, it will delete the image from the docker registry on destroy operation. Defaults to false

triggers (Map of String) A map of arbitrary strings that, when changed, will force the docker_registry_image resource to be replaced. This can be used to repush a local image.

7.2.7 Docker Secret

Manages the secrets of a Docker service in a swarm. It supports the following configurations.

data (String, Sensitive) Base64-url-safe-encoded secret data

name (String) User-defined name of the secret

labels (Block Set) User-defined key/value metadata

7.2.8 Docker Service

Manages the lifecycle of a Docker service. By default, the creation, update and delete of services are detached. Supports the following configurations.

auth (Block List, Max: 1) Configuration for the authentication for pulling the images of the service

converge_config (Block List, Max: 1) A configuration to ensure that a service converges aka reaches the desired that of all task up and running

endpoint_spec (Block List, Max: 1) Properties that can be configured to access and load balance a service

labels (Block Set) User-defined key/value metadata

mode (Block List, Max: 1) Scheduling mode for the service

rollback_config (Block List, Max: 1) Specification for the rollback strategy of the service

update_config (Block List, Max: 1) Specification for the update strategy of the service

7.2.9 Docker Tag

It has the ability to assign user-defined tags to Docker resources. It supports the following configurations.

source_image (String) Name of the source image.

target_image (String) Name of the target image.

id (String) The ID of this resource.

source_image_id (String) ImageID of the source image in the format of sha256:<<ID>>

7.2.10 Docker Volume

It has the ability to define the connection and usage of containers or services running on Docker to a certain disk space. The following configurations can be used.

driver (String) Driver type for the volume. Defaults to local.

driver_opts (Map of String) Options specific to the driver.

labels (Block Set) User-defined key/value metadata.

name (String) The name of the Docker volume (will be generated if not provided).

Along with all these configuration features of Docker, Narwhal offers support for local or remote servers. Similarly, Narwhal supports creating EC2 instances in the AWS Cloud environment and meeting some basic network requirements.

8. CONCLUSIONS

Visual programming languages play a critical role in reducing complexity and improving user experience in modern software development processes. These languages abstract complex systems, making them more accessible and easier to manage. In this context, Narwhal integrates the advantages of visual programming languages into DevOps processes and establishes a direct connection with powerful infrastructure management tools such as Terraform and Terraform CDKTF. With the visual interface offered by Narwhal, users can design infrastructure configurations via diagrams and convert these designs into C# codes with the source-to-source compilation method, making the power of Terraform more accessible through a visual abstraction.

Narwhal's visual abstraction approach not only simplifies infrastructure management, but also prevents common problems such as lack of documentation. Lack of documentation in software projects makes it difficult to share information across teams, making it one of the main causes of architectural erosion. Narwhal effectively addresses this problem by making visual representations a natural part of the documentation. Visualized infrastructure configurations strengthen communication between teams and provide a more shareable and sustainable structure. This is critical, especially in terms of minimizing knowledge loss due to personnel changes in software teams.

Another important contribution of Narwhal is that it makes infrastructure management processes faster and more error-tolerant. Instead of complex infrastructure configuration files prepared with traditional methods, more understandable and accessible solutions can be developed using the visual interfaces offered by Narwhal. Visual representations reduce the possibility of errors in the infrastructure management process and allow for rapid implementation of changes. This not only increases efficiency in software projects, but also ensures long-term sustainability. Narwhal enables users to manage their software infrastructures more effectively, while also enabling projects to be developed in a more dynamic and innovative way.

Such advantages provided by visual programming languages in the field of software development make it possible to evaluate Narwhal's contributions to DevOps processes from a broader perspective. Narwhal aims to standardize infrastructure management processes by providing a common visual template interface for software architecture. This standardization not only accelerates the development processes of projects, but also has the potential to prevent architectural erosion. Integrating Narwhal into popular technologies such as Kubernetes in the future can take these goals to a further level and enable software development teams to manage their business processes more effectively.

As a result, the visual programming language approach offered by Narwhal adds a new dimension to DevOps processes and offers effective solutions to the complexity and sustainability problems encountered in software projects. The convenience provided by visual representations reduces error rates in infrastructure management and strengthens communication and collaboration between teams. As Narwhal supports more technologies in the future and expands its areas of use, it is expected to be more widely adopted in the infrastructure management processes of software projects. In this context, Narwhal has the potential to become an important standard in terms of sustainability and efficiency in the software world.

Further research is needed on how these advantages provided by Narwhal will not only overcome current challenges in software architectures but also shape future software processes.

In this context, there is a need for research that will inspire academic and industrial studies in order to support Narwhal and similar visual programming tools and to provide more comprehensive solutions to the needs of the software community. Studies on visualization and compilation-oriented approaches in software architectures will not only support technical innovations but also make a long-term contribution to the software development culture.

REFERENCES

[1] Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2022). Understanding software architecture erosion: A systematic mapping study. Journal of Software: Evolution and Process, 34(3), e2423.

[2] Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2021, May). Understanding architecture erosion: The practitioners' perceptive. In 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (pp. 311-322). IEEE.

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A systematic mapping study in microservice architecture. 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016 (2016), 44–51.

[4] Chiari, M., Nitto, E. D., Mucientes, A. N., & Xiang, B. (2022, March). Developing a New DevOps Modelling Language to Support the Creation of Infrastructure as Code. In European Conference on Service-Oriented and Cloud Computing (pp. 88-93). Cham: Springer Nature Switzerland.

[5] Piedade, B., Dias, J. P., & Correia, F. F. (2020, October). An empirical study on visual programming docker compose configurations. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (pp. 1-10).

[6] Piedade, B., Dias, J. P., & Correia, F. F. (2022). Visual notations in container orchestrations: an empirical study with Docker Compose. Software and Systems Modeling, 21(5), 1983-2005.

[7] Brikman, Y. (2022). Terraform: Up and Running. " O'Reilly Media, Inc.".

[8] Velepucha, V., & Flores, P. (2023). A survey on microservices architecture:Principles, patterns and migration challenges. IEEE Access.

[9] Stümpfle, J., Jazdi, N., & Weyrich, M. (2024). Tackling Erosion in Variant Rich Software Systems: Challenges and Approaches. Procedia CIRP, 128, 633-637.

[10] Ponce, F., Márquez, G., & Astudillo, H. (2019, November). Migrating from monolithic architecture to microservices: A Rapid Review. In 2019 38th International Conference of the Chilean Computer Science Society (SCCC) (pp. 1-7). IEEE.

[11] Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020). From monolithic systems to microservices: A comparative study of performance. Applied sciences, 10(17), 5797.

[12] De Lauretis, L. (2019, October). From monolithic architecture to microservices architecture. In 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 93-96). IEEE.

[13] Ng, T., Bin Rawi, A. A., Sum, C. S., Tso, E., Yau, P. C., & Wong, D. (2024, February). Migrating from Monolithic to Microservices with Hybrid Database Design Architecture. In Proceedings of the 2024 9th International Conference on Intelligent Information Technology (pp. 536-541).

[14] Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access, 10, 20357-20374.

[15] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. Present and ulterior software engineering, 195-216.

[16] Richardson, C. (2019, March 26). Pattern : Saga. microservices.io. https://microservices.io/

[17] Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). Data management in microservices: State of the practice, challenges, and research directions. arXiv preprint arXiv:2103.00170.

[18] Mosleh, M., Dalili, K., & Heydari, B. (2016). Distributed or monolithic? A computational architecture decision framework. IEEE Systems journal, 12(1), 125-136.

[19] Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., ... & Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. Information and software technology, 131, 106449.

[20] Newman, S. (2021). Building microservices. " O'Reilly Media, Inc.".

[21] Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software, 182, 111061.

[22] Prasandy, T., Murad, D. F., & Darwis, T. (2020, August). Migrating application from monolith to microservices. In 2020 International Conference on Information Management and Technology (ICIMTech) (pp. 726-731). IEEE.

[23] Velepucha, V., & Flores, P. (2021, March). Monoliths to microservices-migration problems and challenges: A SMS. In 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST) (pp. 135-142). IEEE.

[24] Battina, D. S. (2021). The Challenges and Mitigation Strategies of Using DevOps during Software Development. International Journal of Creative Research Thoughts (IJCRT), ISSN, 2320-2882.

[25] Romani, Y., Tibermacine, O., & Tibermacine, C. (2022, March). Towards migrating legacy software systems to microservice-based architectures: a data-centric process for microservice identification. In 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C) (pp. 15-19). IEEE

[26] Kumar, S. S., & Shastry, P. M. (2017, August). Database-per-service for e-learning system with micro-service architecture. In 2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon) (pp. 705-708). IEEE.

[27] Khan, M. S., Khan, A. W., Khan, F., Khan, M. A., & Whangbo, T. K. (2022). Critical challenges to adopt DevOps culture in software organizations: A systematic review. Ieee Access, 10, 14339-14349.

[28] Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison-Wesley Professional.

[29] Woods, E., Erder, M., & Pureur, P. (2021). Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps. Addison-Wesley Professional.

[30] Wolff, E. (2016). Microservices: flexible software architecture. Addison-Wesley Professional.

[31] De Silva, M., & Perera, I. (2015, December). Preventing software architecture erosion through static architecture conformance checking. In 2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS) (pp. 43-48). IEEE.

[32] Chen, L. (2018, April). Microservices: architecting for continuous delivery and DevOps. In 2018 IEEE International conference on software architecture (ICSA) (pp. 39-397). IEEE.

[33] Rios, N., de Mendonça Neto, M. G., & Spínola, R. O. (2018). A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Information and Software Technology, 102, 117-145.

[34] Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015, August). Measure it? manage it? ignore it? software practitioners and technical debt. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 50-60).

[35] Schreiber, A., Nafeie, L., Baranowski, A., Seipel, P., & Misiak, M. (2019, March). Visualization of software architectures in virtual reality and augmented reality. In 2019 IEEE Aerospace Conference (pp. 1-12). IEEE.

[36] Morris, K. (2020). Infrastructure as code. O'Reilly Media.

[37] Morris, K. (2016). Infrastructure as code: managing servers in the cloud. " O'Reilly Media, Inc.".

[38] Bayramcavus, A., Kaya, M. C., & Dogru, A. H. (2021, November). Interoperability of Microservice-Based Systems. In 2021 13th International Conference on Electrical and Electronics Engineering (ELECO) (pp. 594-598). IEEE.

[39] Andrawos, M., & Helmich, M. (2017). Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go. Packt Publishing Ltd.

[40] Guerriero, M., Garriga, M., Tamburri, D. A., & Palomba, F. (2019, September). Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In 2019 IEEE International conference on software maintenance and evolution (ICSME) (pp. 580-589). IEEE.

[41] D. Sokolowski and G. Salvaneschi, "Towards Reliable Infrastructure as Code," 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), L'Aquila, Italy, 2023, pp. 318-321, doi: 10.1109/ICSA-C57050.2023.00072.

[42] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin and D. A. Tamburri, "Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach," 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 2018, pp. 156-15609, doi: 10.1109/ICSA.2018.00025.

[43] G. Rong, Z. Jin, H. Zhang, Y. Zhang, W. Ye and D. Shao, "DevDocOps: Towards Automated Documentation for DevOps," 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 2019, pp. 243-252, doi: 10.1109/ICSE-SEIP.2019.00034.

[44] HashiCorp. (n.d.). Terraform. HashiCorp. https://www.terraform.io/

[45] Stroustrup, B. (n.d.). Interviews and public presentations. Bjarne Stroustrup's homepage. https://www.stroustrup.com/interviews.html

[46] Valcasara, N. (2015). Unreal engine game development blueprints. Packt Publishing Ltd.

[47] Romero, M., & Sewell, B. (2019). Blueprints visual scripting for Unreal engine: The faster way to build games using UE4 blueprints. Packt Publishing Ltd.

[48] Strodick, K., & Schattkowsky, T. (2024, April). Visual Scripting in Unity: A Comparative Analysis of Existing Frameworks. In Proceedings of the ACM/IEEE 8th International Workshop on Games and Software Engineering (pp. 44-49).

[49] J. Sandobalín, E. Insfran and S. Abrahão, "On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven Versus Code-Centric," in IEEE Access, vol. 8, pp. 17734-17761, 2020, doi: 10.1109/ACCESS.2020.2966597.

[50] Ray, P. P. (2017). A survey on visual programming languages in internet of things. Scientific Programming, 2017(1), 1231430.

[51] M. A. Kuhail, S. Farooq, R. Hammad and M. Bahja, "Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review," in IEEE Access, vol. 9, pp. 14181-14202, 2021, doi: 10.1109/ACCESS.2021.3051043.

[52] Noone, M., & Mooney, A. (2018). Visual and textual programming languages: a systematic review of the literature. Journal of Computers in Education, 5, 149-174.

[53] Kulkarni, R., Chavan, A., & Hardikar, A. (2015). Transpiler and it's advantages. International Journal of Computer Science and Information Technologies, 6(2), 1629-1631.

[54] Lattner, C. (2006, April). Introduction to the LLVM compiler infrastructure. In Itanium conference and expo.

[55] Lattner, C., & Adve, V. (2005). The llvm compiler framework and infrastructure tutorial. In Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers 17 (pp. 15-16). Springer Berlin Heidelberg.

[56] Parr, T., & Fisher, K. (2011). LL (*) the foundation of the ANTLR parser generator. ACM Sigplan Notices, 46(6), 425-436.

[57] Parr, T., Wells, P., Klaren, R., Craymer, L., Coker, J., Stanchfield, S., ... & Flack, C. (2004). What's ANTLR.

[58] Rebaudengo, M., Reorda, M. S., Violante, M., & Torchiano, M. (2001, November). A source-to-source compiler for generating dependable software. In Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation (pp. 33-42). IEEE.

[59] Bastidas Fuertes, A., Pérez, M., & Meza Hormaza, J. (2023). Transpilers: A systematic mapping review of their usage in research and industry. Applied Sciences, 13(6), 3667.

[60] Renger, S. (2022). Investigation into the criteria of embeddability of visual scripting languages within the domain of game development.