International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

Revitalizing Legacy Systems: Extracting Key Features for Software Transplantation

[1]Gurjot Singh Sodhi, [2]Dhavleesh Rattan
[1]Research Scholar, Department of Computer Science and Engineering
Punjabi University
Patiala, Punjab, India
e-mail: er.gurjotsinghsodhi@gmail.com
[2]Assistant Professor, Department of Computer Science and Engineering
Punjabi University
Patiala, Punjab, India
e-mail: dhavleesh@gmail.com

Abstract—The creation of intelligent software depends on the ability to transfer software without any restrictions. In this article, a crucial stage in software engineering, the feature extraction for effective software transplantation, is discussed. As hardware, operating systems, or other factors change, it is commonly necessary to move software from one environment to another. It is vital to identify and extract the relevant software characteristics, which might be challenging given how complex software is, in order to carry out efficient software transplantation. On the other hand, the procedure to extract these attributes from the software might be time-consuming and need extensive understanding. To address this, we propose a transplantation strategy that prioritizes automation with the help of AWS. Our approach involves an agent running on the application server (on-premises). It performs the task of feature identification, extraction and deployment on AWS Cloud. Currently, our strategy is confined to Java and .NET applications.

Keywords- Feature, Feature extraction, Intelligent Systems, Microservices, Microservice Architecture, Monolithic Architecture, Transplantation.

I. Introduction

As the software sector grows, more and more people pick up coding and writing code. The same code is typically present in other previously developed software that may be used here when people construct a function in new software. Although the idea of automatically transplanting code is intriguing, current research is mostly focused on experiments that have been carefully planned.

Each software has similar features and functional cores independent of the manufacturer. This observation suggests a similarity to deoxyribose nucleic acid (DNA). Each of us has a unique appearance, but we are all human beings. We have the same ancestor that inherited DNA [21, 22]. Therefore, software transplantation can be defined as:

"*The process of extracting functionality (of interest) from one software/system and deploying it into another (unrelated foreign system) without extensive modifications, making it fully executable, thus minimizing — redesigning, reimplementing and reinventing efforts required for building them from scratch*" [3].

The idea of "*code transplantation*" was coined by Mark Harman and his colleagues in 2015 [1]. Drawing inspiration from the medical field, they likened software to the human body, code to organs, and the process of transplanting code to an organ transplant. To facilitate this process, they introduced the "*μScalpel*" tool, which can automatically transfer a feature from one program to another. Additionally, they introduced novel concepts such as the Donor (the software being transplanted), Organ (the software that can be reused), and Host (the software that enables the reuse of the organ).

This process of transplantation — of functions or attributes between software can save human programmers from cumbersome standard work and make developing software faster and cheaper [23]. Software Transplantation basically make use of Abstraction[1] and Refinement[2] concepts as shown in Figure 1. Transplantation approach is based on Genetic Improvement (GI), which treats the code as 'genetic material' that can be manipulated to improve the system. GI can repair broken functionalities, drastically scale-up performance, and port between dialects and platforms. This is a program synthesis that has recently become the subject of much activities.

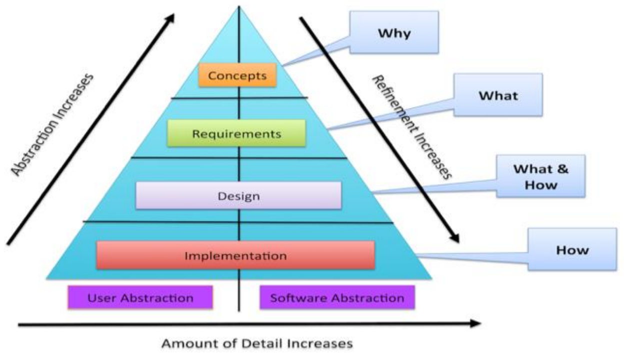


Figure 1. Abstraction and Refinement in Context to Software [3]

[1] By hiding information about other features, the abstraction focuses on selected system features.

[2] Refinement focuses on supplanting certain aspects of the system with some insights.

A. Need for Software Transplantation

Software Transplantation is a long-term development practice aimed at multiple disciplines. In the event that if the recipient already has features that needs to be transplanted, we don't perform software transplantation. If so, at that point we need to strip the current functionalities from the recipient. Figure 2 portrays the need for software transplantation. Transplantation will broaden new horizons for upgrading software development practices [3].

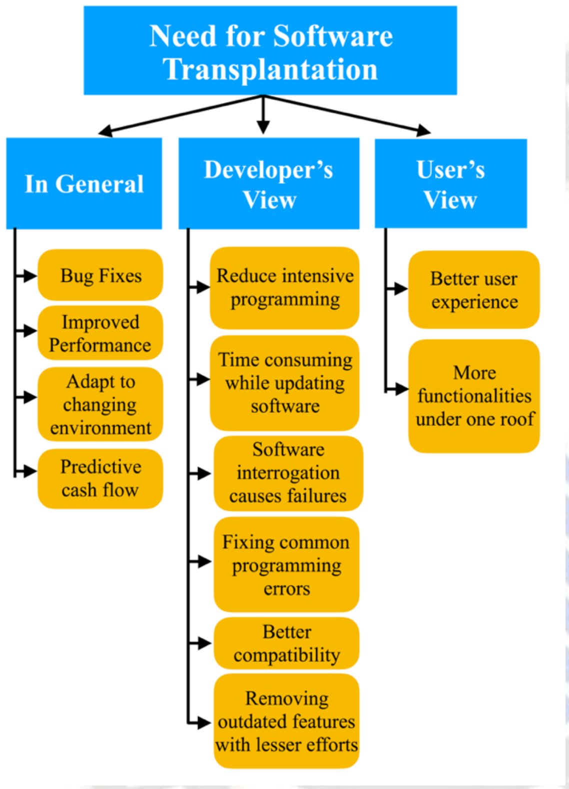

Need for Software Transplantation
In General
Developer's View
User's View
Bug Fixes
Improved Performance
Adapt to changing environment
Predictive cash flow
Reduce intensive programming
Time consuming while updating software
Software interrogation causes failures
Fixing common programming errors
Better compatibility
Removing outdated features with lesser efforts
Better user experience
More functionalities under one roof

Figure 2. Need for Software Transplantation Software [3]

This approach can be incredibly useful under the accompanying circumstances:

1. When we don't have a system with the necessary computation power.
2. When we are satisfied with the outcomes accomplished by any exemplar trained on DONOR system, and we would prefer not to re-train it in the HOST system.
3. When it takes enormous time to train any network on HOST system.

Employing the proposed technique, we can extricate the exemplar from the well-trained DONOR and offer them to the HOST getting the same performance across the subject systems.

B. Software Features

Software features are features that consumers and developers may both utilise. Software frameworks' source codes are modified by developers to include the newest features [2], enhance built-in functionality, and get rid of outmoded ones. When programmers are assigned with modifying the origin code of a considerable or unknown structure, they dedicate a significant amount of time and effort in program understanding tasks to acquire the expertise essential to implement the modifications. A component of this strategy is referred to as "*feature location*", an activity in software development whereby programmers look for entities in the source code (such as methods or classes) which implement features [3].

It interprets each component of the source code based on Structural analysis[3], Behavioral analysis[4] or Semantic analysis[5] when it comes to information acquisition akin to source code [3]. Table 1 demonstrates how to submit such information as needed.

TABLE I. INFORMATION TO BE CAPTURED FROM SOURCE CODE

What we get	Analysis method		
	Behaviora l Analysis	Structura l Analysis	Seman tic Analysis
Annotations	✓	✓	✓
Regular Expressions	✓	✓	✓
Classes	✓	✓	✓
Attributes	✓	✓	✓
Parameters	✓	✓	✓
Call Actions	✓	✓	✓
Dependencies	✓	✓	✓
Relationship with other classes	✓	✓	✓
Methods	✓	✓	✓
Structures	✓	✓	✓
Return types	✓	✓	✓
Inheritance	✓	✓	✓
Object Instantiations	✓	✓	✓

C. Microservices

In recent times, the field of software engineering has witnessed a growing trend towards cloud computing [4]. As the infrastructure landscape evolves, there is a growing need for architectural styles that can effectively harness the opportunities provided by cloud infrastructure while addressing the complexities involved in developing cloud-native applications.

One architectural style that has garnered significant attention in the industry within this context is the microservices architecture. This approach has been extensively discussed and explored in various sources [5, 6, 7, 8]. For many years, software companies have relied on monolithic enterprise applications as their preferred architecture. This approach worked well within a limited scope and manageable support requirements, but challenges arose as systems grew in size and complexity. In response, the approach of breaking down monolithic applications into smaller, autonomous microservices emerged. Each microservice can be deployed and maintained by an agile team of software engineers, eliminating the need for extensive cross-team collaboration.

Microservices architecture (MSA) provides a solution to the issues associated with traditional monolithic backend applications. However, simply dividing applications into containers does not automatically guarantee scalability. It is

[3] The investigation of inter-class connections, method calls, and data types constitutes structural analysis.

[4] Behavioral analysis methodologies revolve around the program's execution behavior.

[5] Semantic analysis provides a supplementary perspective to the structural and behavioral aspects.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

crucial to undertake proper planning to ensure effective execution and coordination among these individual containers. The objective of this work is to thoroughly examine and establish a formal framework for addressing the key challenges associated with extracting and transplanting specific features of interest from one application (DONOR) to another application (HOST).

The available literature indicates that the term "*microservice*" lacks a formal definition [7, 8]. According to Sam Newman [9], microservices are small, autonomous services that collaborate to perform a specific task proficiently. This implies that each microservice focuses on a concise, well-defined section of the problem domain. Eric Evans introduced the concept of bounded context, wherein a bounded context contains domain entities that are relevant only within that context and shares only the necessary entities for communication with other bounded contexts [10]. To adhere to the identified bounded contexts within the domain, the literature on microservices recommends constructing services [7, 9]. This approach enables the development of cohesive and decoupled services that deliver resources or functions specific to their respective bounded contexts.

In the microservice architecture, a suite of microservices collaborate to form a single, large application. These microservices communicate with each other through lightweight mechanisms such as HTTP or remote procedure calls [7]. Essentially, microservices introduce a new form of componentization, where a component is not limited to a class, package, or library, but rather an independently deployable service that operates in its own processes. This architecture takes the principle of loose coupling and high cohesion to the extreme (Jong Kook Lee et al., 2001). As a result, communication mechanisms are kept lightweight and devoid of business logic, often referred to as "smart endpoints and dumb pipes" [7].

Each microservice in a microservices architecture is comparable to a station in an assembly line for manufacturing. Microservices function in a similar way to stations, where each is in charge of a single job (Figure 3). Each station or microservice is a specialist in its specific area of responsibility, which promotes productivity, consistency, and output quality. Compare it to a production setting where each station is in charge of constructing the complete product alone. That is comparable to a software programme that executes all operations through a single procedure.

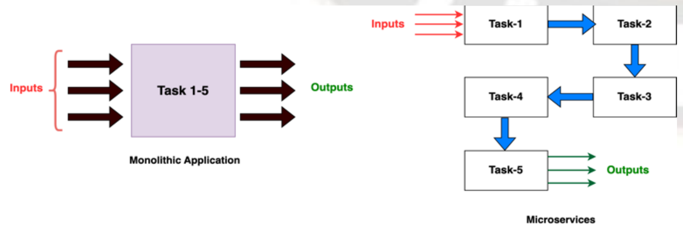


Figure 3. Monolithic and Microservice Architectures

To be clear, since assembly lines and microservices do not always operate in a strictly serialised manner, the assembly line comparison does not imply a single linear flow (Figure 4). Microservices make it simple to copy data, distribute it to various locations as part of the data pipeline, and then process it in various ways as in a directed acyclic graph (DAG). This allows you more freedom in how you build the data pipeline and makes it easier to expand it should you decide to add more outputs to the flow.

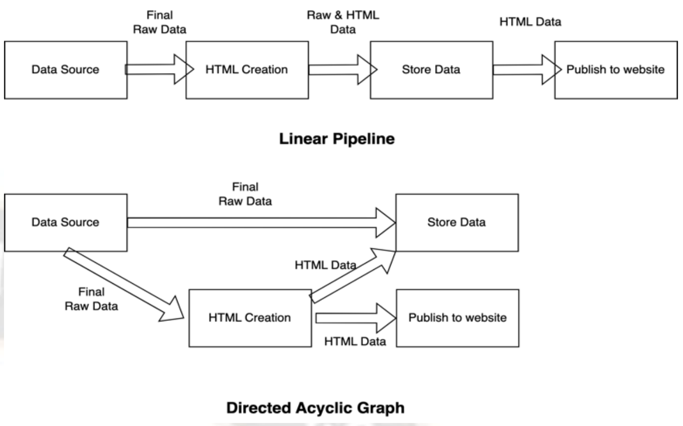


Figure 4. Data Flow in a Microservices Architecture

D. Identify/Discover Features

The task of identifying the crucial operations carried out by the programme is known as feature discovery in software designing [24, 25]. Features are characterized as client-centric marques that specify how a programme behaves (as an example, "plays mp3 files") [26]. Programmes are frequently seen of as implementing a collection of features, hence the idea of features is important. Through requirements elicitation and domain analysis [27, 28], software developers identify the features that need to be implemented. With the use of traceability, engineers combine feature representations with different software relics [29]. The software has passed the requisite feature set actualization test, which is proof of its functionality [30]. For security or privacy concerns, regulatory regulations frequently demand that certain features be included [31]. As a result, understanding the features that are really implemented by a particular piece of software is a crucial task. Figure 5 illustrates the taxonomy of feature discovery.Identify/Discover Features

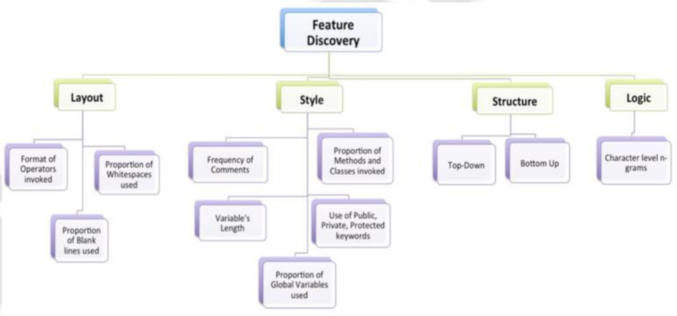

Feature Discovery
Layout
Style
Structure
Logic
Format of Operators invoked
Proportion of Whitespaces used
Proportion of Blank lines used
Frequency of Comments
Proportion of Methods and Classes invoked
Variable's Length
Use of Public, Private, Protected keywords
Proportion of Global Variables used
Top-Down
Bottom Up
Character level n-grams

Figure 5. Taxonomy for Feature Discovery [3]

E. Extract Features

By performing feature extraction, the problem of selecting the most concise and useful set of features is solved. Each characteristic or parameter of a feature is produced through either a quantitative or qualitative assessment [12]. Fig 6 illustrates the interaction among common components such as a package, class, method, attribute, and source file, which are the features that require extraction.

1. A package organizes homogenous classes and associated interfaces.
2. Apiece source file is an individual compilation component connected with a singular package. It defines atlas either none or multiple classes or associated interfaces.
3. Classes subsist containers for assembling functions and features of interconnected variables and operations.
4. Methods being mere segments of code that furnish a sequence of directives or declarations for carrying out a specified task.
5. An attribute being a variable or unchanging value that pertains to a class or its associated interfaces.

Abstract Syntax Trees (ASTs): These source code representations, which resemble trees, encapsulate the grammar and organization of the code. They may be used to glean information about variables, function calls, and connections between different areas of the code.

Dependency Analysis: Tools for dependency analysis look at the connections between various code components and create dependency graphs. This can help identify which code portions are interconnected and how changes to one region of the code may affect other areas.

Detecting Code Clones: In a codebase, duplicated code fragments are found using code clones detection techniques. This can be useful for identifying hidden dependencies between various parts of the code.

Source Code Repository Mining: To learn about code changes over time, source code repository mining examines a codebase's version control history. This can help with understanding the connections between different parts of the code and how the code has evolved over time.

Program Slicing: A technique for isolating a portion of code related to a given functionality or feature is known as programme slicing. This can aid in discovering relationships between various areas of the code and comprehending how they contribute to overall functioning.

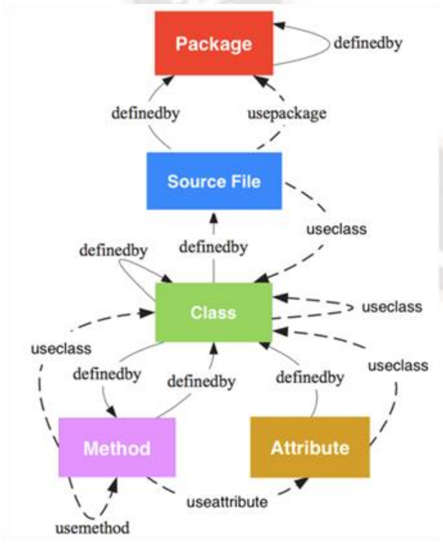


Figure 6. Package, Class, Method, and Attribute Interconnectivity [3]

Our work makes the following contributions:

1. Minimize the amount of time a programmer spends identifying/locating/extracting application-of-interest from the application server;
2. Assist the programmer in analyzing code; and
3. Assist the programmer in transplanting organs.

II. Related Work

We explore the existing literature on software transplantation, microservices, as well as examine prior research in traditional software engineering disciplines that offer relevant methodologies and techniques. These disciplines encompass areas such as reverse engineering, system decomposition and maintenance, which provide valuable insights for understanding and applying microservice principles. By considering these diverse fields, we can draw upon a comprehensive body of knowledge to enhance our understanding and approach to microservices.

A. Software Transplantation

Wenyong [13] provided a comprehensive overview of decompilers, optimal reduction, control flow, and data flow analysis on Micro-VAX II, VMS4.4 operating system using organizational analysis and C language feature recovery.

Poe and co-workers [14] developed a method for generating feature-based parameterized value-based (transactional) memory benchmarks. TransPlant, a benchmark developed by authors can generate parameterized, complete value-based workloads naturally using decentralized source-code.

Haitao [15] proposed an improved software feature similarity disposal technique based on the k-means clustering algorithm that ensure effective software transplantation.

Barr and Harman [1] put forward a theory, a tool (µScalpel), and an algorithm (µTrans) that integrate static analysis and dynamic analysis to extract, modify, and transplant code from a donor system into a host.

A. Marginean in [16] employs a lightweight annotation framework in conjunction with a Search Based approach (augmented by static analysis) to automatically transplant missing features from Kate using the tool Scalpel provided by [1].

Dash et al. [17] concentrate on the Linux kernel, OMAP-L138, a Linux Kernel configurability, the workflow of transplanting U-Boot and the Linux Kernel to the OMAP-L138, and cross-compiling the Linux kernel to generate architecture-specific code. GRAFTER, a test transplantation-reuse strategy that enables runtime behaviour-correlation among clones, was introduced by Zhang [18].

Petke [19] use genetic development to build a faster variety of a C++ software called MiniSAT, that being a Boolean satisfiability solver; incorporating image processing tools ImageMagick and GraphicsMagick.

Liu [20] demonstrates that program slicing can handle features such as unknown source, irregular, and ambiguous function description in the context of open source software that allows code transplantation.

Wang and co-workers [21] perform an empirical study on organ removal from the GitHub repository to investigate transplantation dependence on large-scale datasets for specific platforms.

Sodhi and Rattan [3] conducted a systematic research demonstrating the necessity for software transplantation as well as a comprehensive examination of how it should be carried out.

B. Microservices

Microservices architecture has gained significant popularity in recent years due to its ability to address the challenges posed by monolithic applications. Researchers and software developers have been actively working on various approaches to decompose monolithic applications into microservices. One major challenge in decomposing monolith applications is the lack of tools and clear measures to evaluate the quality of the decomposed systems. Several research studies have focused on addressing this challenge and providing guidelines for the successful decomposition of monolith applications into microservices.

The field of microservices is relatively young and lacks a substantial body of research. Pahl and Jamshidi conducted a systematic secondary study to review and classify existing research on microservices [38]. Their study focused on 21 research papers published in 2014 and 2015, making it the first of its kind for microservices. The findings indicate that the research on microservices is still in an immature and formative stage. Moreover, the review highlights the need for more experimental evaluation of proposed solutions and their benefits within the microservices research community. The study also reveals a lack of tool support for microservices in the current state of the art.

Taibi and others [33] presents a comprehensive review of literature on microservices and identifies key research trends and challenges in the field. It explores various aspects of microservices, including decomposition techniques, communication protocols, deployment strategies, and monitoring approaches. The paper also highlights the need for more empirical studies and guidelines to aid practitioners in successfully adopting microservices.

Ayas H.M. et al.'s qualitative research [34] analyses 215 StackOverflow conversations and 19 interviews to present an overview of how microservice migrations occur as well as a breakdown of high level modes of change to particular solution results.

In order to map the existing microservices-specific techniques and understand how to continuously deliver value in a DevOps pipeline, Taibi D. et al.'s conducted a systematic mapping study [35] characterising the various microservice architectural style principles and patterns.

In order to demonstrate the knowledge and significance of the Microservice architecture (MSA), a systematic mapping study (SMS) was carried out by [36]. In order to identify trends, obstacles, successful variables, and possible industrial adoption connected to microservice architecture, the authors stress the importance of MSA, the necessity for thorough research on migration methodologies, and the conclusions of the systematic mapping study.

In a study published in 2023, Hamza M. looked into the necessity of moving from monolithic to microservices architecture, as well as the architectural description for doing so, refactoring tools, methods, and potential challenges and strategies while successfully converting to microservices [37].

III. PROPOSED METHODOLOGY

As per Amazon Web Services (AWS), *Application Modernization* is the process to create new business value from existing application environments by updating them with modern features and capabilities. By modernizing your legacy application environments, you can include the latest functionalities that better align with what your business needs to succeed.

A. Problem Statement

There is no single path that our customers take when they modernize on AWS, but once you start modernizing your applications you get many important benefits like below:

1. Agility: Develop and deploy faster to achieve business goals.
2. Enterprise DevOps: Build and operate utilizing proven ecosystem of cloud-native tooling.
3. Portable & Isolated: Enable portable, scalable and isolated application deployment.
4. Operational Efficiency: Reduce IT operational overhead and achieve optimized compute infrastructure.

But you have few challenges:

1. The applications are old and the application owner had left the company years ago without leaving any documentation behind.
2. You/your team is not expert on the containerization process, thus, you are not feeling comfortable with touching these applications.
3. You/your team has recently started your Cloud Journey with AWS and you want to be sure whether you are following the Cloud Best practices while deploying the applications into the Cloud.
4. You/your team is very busy and can't spend too much time to containerize these applications.

B. Decomposing applications into services

The scale cube, which is depicted in Figure 7 below, is a really helpful 3-D scalability model that is described in the book The Art of Scalability [32].

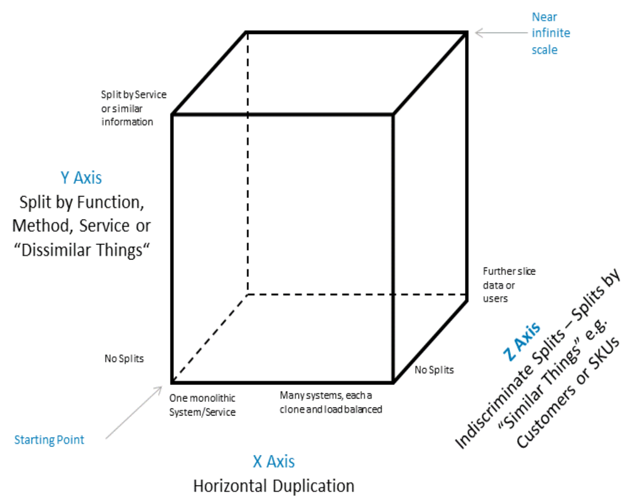

Near infinite scale
Split by Service or similar information
Y Axis
Split by Function, Method, Service or "Dissimilar Things"
Further slice data or users
No Splits
No Splits
One monolithic System/Service
Many systems, each a clone and load balanced
Starting Point
X Axis
Horizontal Duplication
Z Axis
Indiscriminate Splits – Splits by "Similar Things" e.g. Customers or SKUs

Figure 7. The Scale Cube [32]

In this concept, X-axis scaling refers to the widely used method of scaling an application by running several identical copies of the application in front of a load balancer. That's a fantastic approach to increase an application's capacity and accessibility.

Each server runs an identical copy of the code when Z-axis scaling is used. It is comparable to X-axis scaling in this way. The main distinction is that each server is solely in charge of a portion of the data. Each request is sent to the proper server via a system component. An property of the request, such as the main key of the entity being requested, or sharding, is a frequently used routing criterion. The sort of consumer is another regular routing criterion. For instance, an application may route requests from paying users to a different group of servers with additional capacity, giving them a better SLA than those from free users.

The capacity and availability of the application are enhanced by Z-axis scaling, much like with X-axis scaling. However, neither strategy addresses the issues of growing application and development complexity. Applying Y-axis scaling is what we need to do to fix those issues.

Functional decomposition, often known as Y-axis scaling, is the third dimension of scaling. Y-axis scaling divides items that are distinct, whereas Z-axis scaling divides things that are similar. A monolithic application is divided into a number of services at the application tier thanks to Y-axis scalability. Each service implements a group of linked features, such as customer and order management.

C. Methodology

There are many important factors to conceive when transplanting a microservice-based application to the AWS cloud. First, the application needs to be decomposed into smaller microservices that are designed to perform specific tasks or functions. This decomposition requires careful analysis of the existing monolithic application to identify logical boundaries and separate functionalities into distinct services. Once the decomposition is complete, each microservice can be containerized using technologies like Docker or Kubernetes for easy deployment and management on AWS. To enable seamless interaction and interoperability, standardizing communication across microservices is also crucial. Via well-defined API calls, microservices may communicate with one another.

It is vital to take into account the scalability and fault tolerance offered by containerization technologies like Docker and AWS' Elastic Container Service or Elastic Kubernetes Service when transplanting a microservices-based application to the AWS cloud. Additionally, security precautions must be taken to safeguard sensitive data and guarantee conformity with industry regulations. It is crucial to utilize the numerous services and features provided by AWS in order to properly transplant a microservice-based application into the AWS cloud. These include serverless computing solutions like AWS Lambda, AWS Fargate or container orchestration services such as Amazon Elastic Container Service, Elastic Kubernetes Service, as well as security management solutions like AWS Identity and Access Management.

The detailed architecture for the proposed approach is depicted in the Figure 8.

The application should also be built with scalability in mind throughout design and development. This may be done by utilizing auto-scaling tools offered by AWS, such as Application Load Balancers or Amazon EC2 Auto Scaling. Additionally, utilizing AWS's cloud-native technologies like Elastic Load Balancers and auto-scaling groups can aid in ensuring high availability and fault tolerance for the microservices.

The *Anti-corruption Layer* serves as a bridge between bounded contexts, facilitating communication and ensuring that data in each context aligns with its specific language and treatment. It acts as a translator, enabling seamless integration and maintaining consistency between different contexts.

Prerequisites

1. AWS Account with relevant permissions.
2. Remote access to the worker machine on which your application (monolithic application) is currently running.
3. Docker engine configured on the worker machine.
4. Confirm your application(s) falls under the supported applications[6] list.

Requirements

1. Amazon Simple Storage Service (S3) bucket to store your artifacts.
2. Create an AWS Identity and Access Management (IAM) user that has access to the Amazon S3 buckets and a designated Amazon Elastic Container Registry (ECR).
3. Deploy a worker node as an Amazon Elastic Compute Cloud (Amazon EC2) instance. This will include a compatible operating system (Linux/Windows), which will take the artifacts and convert them into containers.
4. Install the Application-Container (A2C) agent[7] on each server (Amazon EC2 instance) that you want to transplant.

Agent analyzes the selected application, packages up its dependencies (for example, open network ports or third-party libraries in use), and generates the relevant container artifacts, such as the container image, task definitions, and YAML[8] files for easy deployment to Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS).

Figure 9 below shows the sequence of steps, with the output of each being leveraged as the inputs to the subsequent steps in the sequence.

[6] Java applications (Linux) or ASP.NET applications (Windows, Linux).

[7] You can save time when containerizing a fleet of machines by automation, using AWS Systems Manager.

[8] YAML stands for yet another markup language.

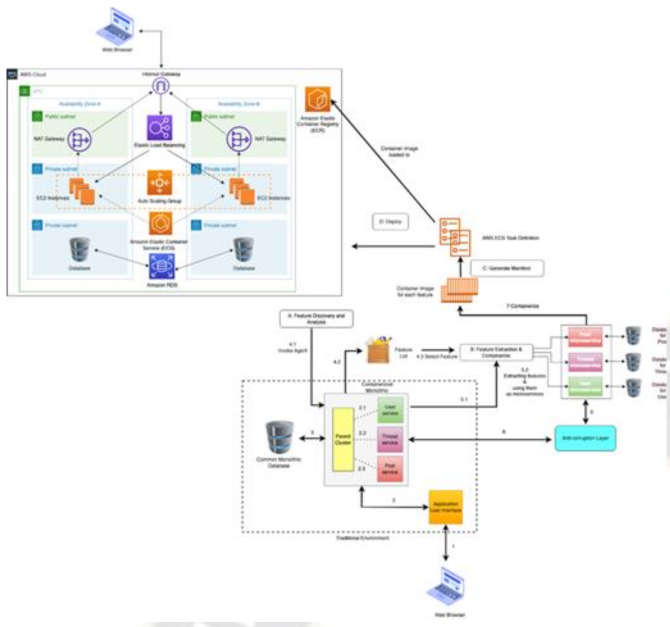

Web Browser
AWS Cloud
Amazon Elastic Container Registry (ECR)
Amazon ECS Task Definition
C: Generate Manifest
D: Deploy
A: Feature Discovery and Analysis
B: Feature Extraction & Containerize
API Gateway
Anti-corruption Layer
Application User Interface
Common Monolithic Database
Traditional Environment
Web Browser

Figure 8. Architecture Diagram for the Proposed Approach

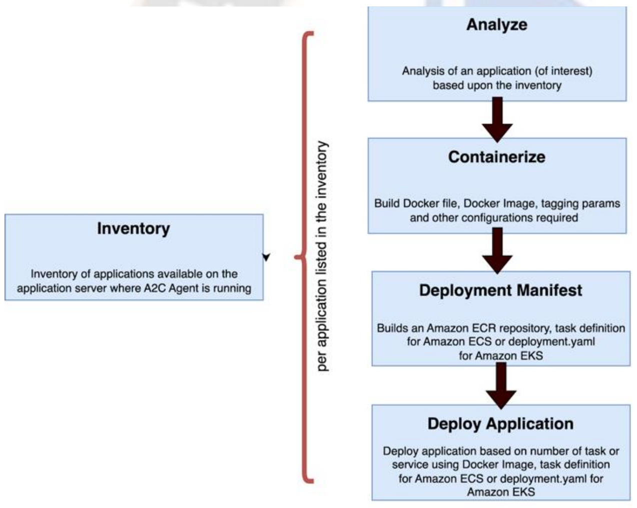

Analyze
Analysis of an application (of interest) based upon the inventory
Containerize
Build Docker file, Docker Image, tagging params and other configurations required
Deployment Manifest
Builds an Amazon ECR repository, task definition for Amazon ECS or deployment.yaml for Amazon EKS
Deploy Application
Deploy application based on number of task or service using Docker Image, task definition for Amazon ECS or deployment.yaml for Amazon EKS
Inventory
Inventory of applications available on the application server where A2C Agent is running
per application listed in the inventory

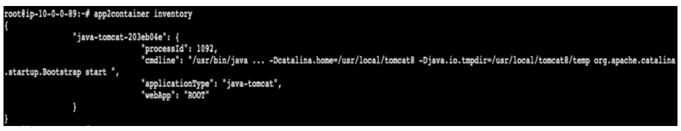
Figure 9. Automation Sequence

IV. RESULTS AND DISCUSSION

A. *Feature Identification/Discovery*

First we need to identify the entry points for our application before we proceed with the transplantation procedure (refer section 1.4). In this case, the agent will –

1. Creates an inventory list (Figure 10) for all applications running in virtual machines, on-premises, or in the cloud which so ever is your use-case and lists them in a JSON format.



```
root@ip-10-0-0-89:~# app2container inventory
{
            "java-tomcat-203eb04e": {
                        "processId": 1092,
                        "cmdline": "/usr/bin/java ... -Dcatalina.home=/usr/local/tomcat8 -Djava.io.tmpdir=/usr/local/tomcat8/temp org.apache.catalina
.startup.Bootstrap start ",
                        "applicationType": "java-tomcat",
                        "webApp": "ROOT"
            }
}
```

Figure 10. Getting Inventory Information from Application Server

2. Analyzes the run-time dependencies (Figure 12) of supported applications that are running, including cooperating processes and network port dependencies.

As you can see in Figure 11, we need to set up an Amazon EC2 instance (virtual server) as the worker node, an S3 bucket for the analysis output, and two AWS Systems Manager documents. The first document is run on the target server. It will install a container agent and run the analysis steps. The second document is run on the worker node and handles the deployment of the container image. Analysis runs on the target server we are transplanting and deployment runs on the worker node.

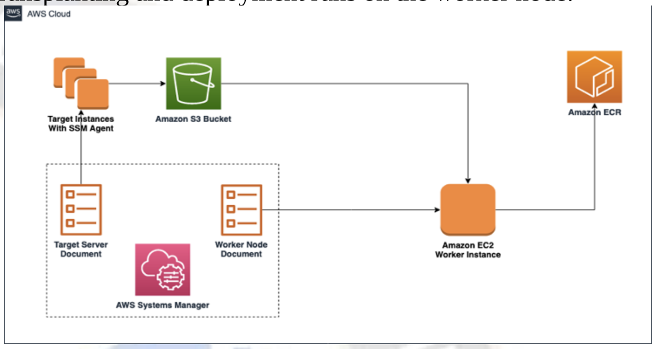
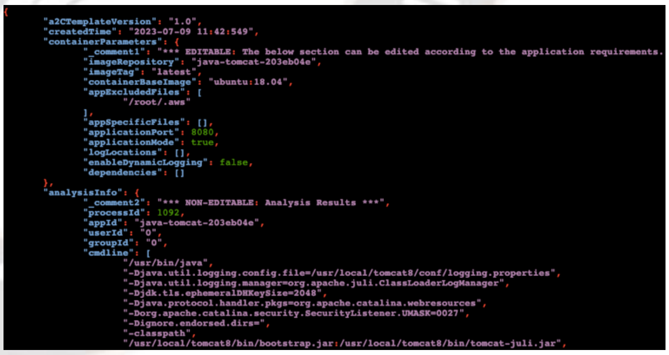


Figure 11. Overview of the Working Architecture Supporting Scalability

There are three important aspects of the generated analysis output:

1. *Issues and observations:* The agent generates a text file (named report.txt) providing a list of any issues and observations found that may need remediating before proceeding for containerization.
2. *Dependency identification:* The agent not only analyzed and gathered details about the applications, but also identified dependencies via its “Co-operating process” capability. A snapshot of the analysis.json depicting this is Figure 12 below.



```
{
        "a2CTemplateVersion": "1.0",
        "createdTime": "2023-07-09 11:42:549",
        "containerParameters": {
                "_comment1": "*** EDITABLE: The below section can be edited according to the application requirements.
                "imageRepository": "java-tomcat-203eb04e",
                "imageTag": "latest",
                "containerBaseImage": "ubuntu:18.04",
                "appExcludedFiles": [
                        "/root/.aws"
                ],
                "appSpecificFiles": [],
                "applicationPort": 8080,
                "applicationMode": true,
                "logLocations": [],
                "enableDynamicLogging": false,
                "dependencies": []
        },
        "analysisInfo": {
                "_comment2": "*** NON-EDITABLE: Analysis Results ***",
                "processId": 1092,
                "appId": "java-tomcat-203eb04e",
                "userId": "0",
                "groupId": "0",
                "cmdline": [
                        "/usr/bin/java",
                        "-Djava.util.logging.config.file=/usr/local/tomcat8/conf/logging.properties",
                        "-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager",
                        "-Djdk.tls.ephemeralDHKeySize=2048",
                        "-Djava.protocol.handler.pkgs=org.apache.catalina.webresources",
                        "-Dorg.apache.catalina.security.SecurityListener.UMASK=0027",
                        "-Dignore.endorsed.dirs=",
                        "-classpath",
                        "/usr/local/tomcat8/bin/bootstrap.jar:/usr/local/tomcat8/bin/tomcat-juli.jar",
```

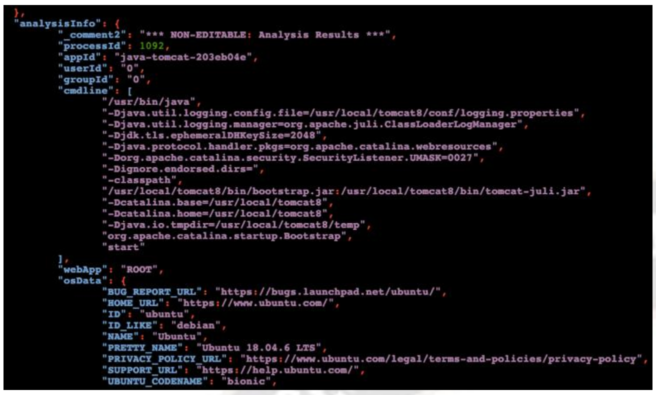
Figure 12. Snippet of Analysis.json with Dependency Information

In the above screenshot, the application in question (with processId 1092 is shown as non-dependent in the analysis report. This analysis.json file also includes a section detailing the dependency found, such as the port used for connectivity, as well as other access parameters. Agent uses this information in subsequent steps to ensure these dependencies (if any) are factored into the generated containers.

3. *Application start-up information:* The deep scanning capabilities supported by the agent also provides an add-on in terms of identifying the start-up command

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9
***Article Received:** 25 July 2023 **Revised:** 12 September 2023 **Accepted:** 30 September 2023*

and information (Figure 13) for each legacy application available on the application server. Agent uses this information in subsequent steps to launch these applications within containers.



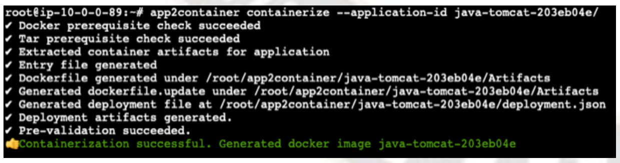
```
},
"analysisInfo": {
        "_comment2": "*** NON-EDITABLE: Analysis Results ***",
        "processId": 1092,
        "appId": "java-tomcat-203eb04e",
        "userId": "0",
        "groupId": "0",
        "cmdline": [
                "/usr/bin/java",
                "-Djava.util.logging.config.file=/usr/local/tomcat8/conf/logging.properties",
                "-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager",
                "-Djdk.tls.ephemeralDHKeySize=2048",
                "-Djava.protocol.handler.pkgs=org.apache.catalina.webresources",
                "-Dorg.apache.catalina.security.SecurityListener.UMASK=0027",
                "-Dignore.endorsed.dirs=",
                "-classpath",
                "/usr/local/tomcat8/bin/bootstrap.jar:/usr/local/tomcat8/bin/tomcat-juli.jar",
                "-Dcatalina.base=/usr/local/tomcat8",
                "-Dcatalina.home=/usr/local/tomcat8",
                "-Djava.io.tmpdir=/usr/local/tomcat8/temp",
                "org.apache.catalina.startup.Bootstrap",
                "start"
        ],
        "webApp": "ROOT",
        "osData": {
                "BUG_REPORT_URL": "https://bugs.launchpad.net/ubuntu/",
                "HOME_URL": "https://www.ubuntu.com/",
                "ID": "ubuntu",
                "ID_LIKE": "debian",
                "NAME": "Ubuntu",
                "PRETTY_NAME": "Ubuntu 18.04.6 LTS",
                "PRIVACY_POLICY_URL": "https://www.ubuntu.com/legal/terms-and-policies/privacy-policy",
                "SUPPORT_URL": "https://help.ubuntu.com/",
                "UBUNTU_CODENAME": "bionic",
```

Figure 13. Analysis.json with Application Start-Up Command

B. *Feature Extraction*

The activities in this stage depend on whether all steps are performed on the application server or whether analysis is performed on the application server and containerization and deployment are performed on a worker machine. In this stage, the agent leveraged all of the application and dependency information from the analysis step (see section 4.1) to generate the container image for the applications. Also, there are three important value add-ons in this stage -

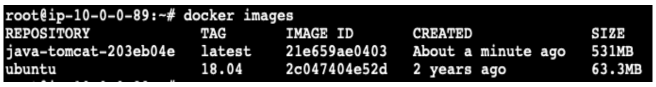
1. *Creation of Dockerfile*: Agent not only created the container image (Figure 14), but also generated the Dockerfile and made it available in an AWS CodeCommit repository that it created for the application to hold all the necessary object files. If the application needs to be changed further later on, this Dockerfile needs to be updated so that it can be reused to regenerate the Docker image for the changed code. This is a great value-add for use cases where the application is modernized further after it's transplanted to the cloud.



```
root@ip-10-0-0-89:~# app2container containerize --application-id java-tomcat-203eb04e/
✓ Docker prerequisite check succeeded
✓ Tar prerequisite check succeeded
✓ Extracted container artifacts for application
✓ Entry file generated
✓ Dockerfile generated under /root/app2container/java-tomcat-203eb04e/Artifacts
✓ Generated dockerfile.update under /root/app2container/java-tomcat-203eb04e/Artifacts
✓ Generated deployment file at /root/app2container/java-tomcat-203eb04e/deployment.json
✓ Deployment artifacts generated.
✓ Pre-validation succeeded.
👍Containerization successful. Generated docker image java-tomcat-203eb04e
```

Figure 14. Initiating the Containerization Process

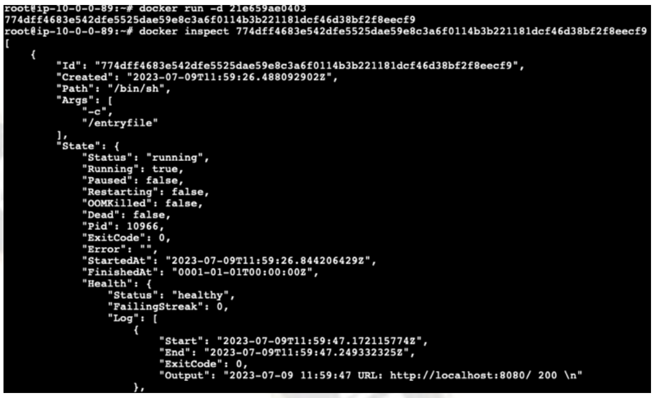
2. *Image tagging*: The Docker file generated automatically applied image tagging and tagged the generated image as latest, thus handling versioning automatically.



```
root@ip-10-0-0-89:~# docker images
REPOSITORY             TAG       IMAGE ID       CREATED              SIZE
java-tomcat-203eb04e   latest    21e659ae0403   About a minute ago   531MB
ubuntu                 18.04     2c047404e52d   2 years ago          63.3MB
```

Figure 15. Tagging The Docker Image

3. *Customization*: Agent automatically made intelligent decisions while generating the Docker file and image, such as selection of the right base image. In case there is a need to make changes to these, such as using a different base image, the generated Dockerfile is available to be edited as needed. You can also test (this will run the container in background) and inspect (this will dump a large amount of information about the container) your docker image (Figure 16).

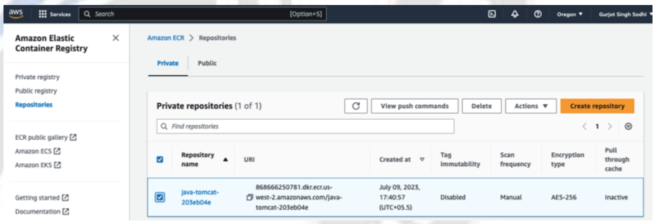


```
root@ip-10-0-0-89:~# docker run -d 21e659ae0403
774dff4683e542dfe5525dae59e8c3a6f0114b3b221181dcf46d38bf2f8eecf9
root@ip-10-0-0-89:~# docker inspect 774dff4683e542dfe5525dae59e8c3a6f0114b3b221181dcf46d38bf2f8eecf9
[
    {
        "Id": "774dff4683e542dfe5525dae59e8c3a6f0114b3b221181dcf46d38bf2f8eecf9",
        "Created": "2023-07-09T11:59:26.488092902Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "/entryfile"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 10966,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2023-07-09T11:59:26.844206429Z",
            "FinishedAt": "0001-01-01T00:00:00Z",
            "Health": {
                "Status": "healthy",
                "FailingStreak": 0,
                "Log": [
                    {
                        "Start": "2023-07-09T11:59:47.172115774Z",
                        "End": "2023-07-09T11:59:47.249332325Z",
                        "ExitCode": 0,
                        "Output": "2023-07-09 11:59:47 URL: http://localhost:8080/ 200 \n"
                    },
```

Figure 16. Testing and Inspecting the Docker Image

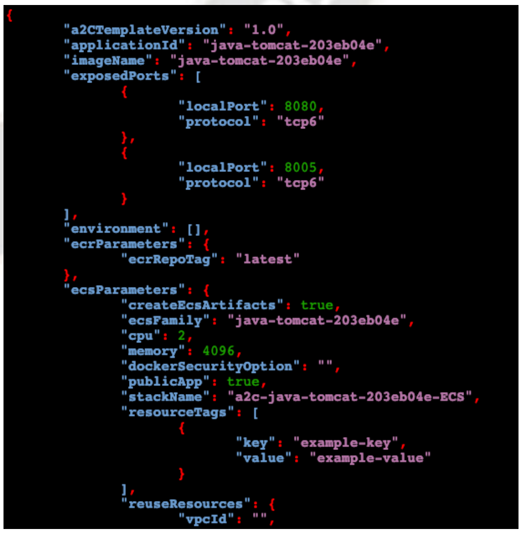
C. *Artifacts for Deployment*

In this stage, the agent will generate the artifacts needed to deploy your application container in AWS. It generates the Amazon ECS task definition (Figure 18) and registers the task to run on the created ECS cluster; transfers the application container image you prepared into an Amazon ECR repository (Figure 17) created by the agent. It then used the Docker images created earlier to launch the applications into ECS as containers.



Repository name	URI	Created at	Tag immutability	Scan frequency	Encryption type	Pull through cache
java-tomcat-203eb04e	868666250781.dkr.ecr.us-west-2.amazonaws.com/java-tomcat-203eb04e	July 09, 2023, 17:40:57 (UTC+05.5)	Disabled	Manual	AES-256	Inactive

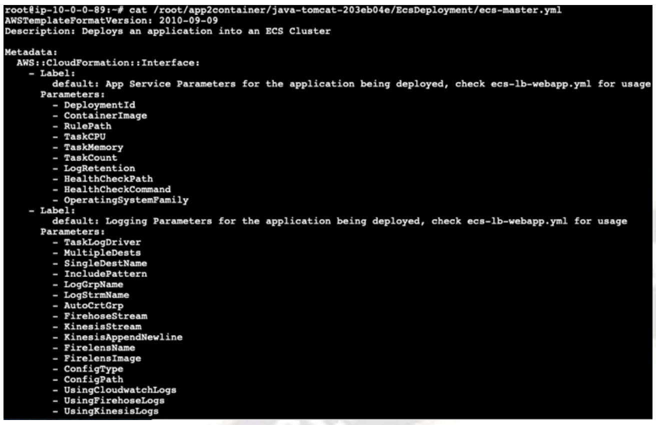
Figure 17. Amazon ECR Repository created by Agent



```
{
        "a2CTemplateVersion": "1.0",
        "applicationId": "java-tomcat-203eb04e",
        "imageName": "java-tomcat-203eb04e",
        "exposedPorts": [
                {
                        "localPort": 8080,
                        "protocol": "tcp6"
                },
                {
                        "localPort": 8005,
                        "protocol": "tcp6"
                }
        ],
        "environment": [],
        "ecrParameters": {
                "ecrRepoTag": "latest"
        },
        "ecsParameters": {
                "createEcsArtifacts": true,
                "ecsFamily": "java-tomcat-203eb04e",
                "cpu": 2,
                "memory": 4096,
                "dockerSecurityOption": "",
                "publicApp": true,
                "stackName": "a2c-java-tomcat-203eb04e-ECS",
                "resourceTags": [
                        {
                                "key": "example-key",
                                "value": "example-value"
                        }
                ],
                "reuseResources": {
                        "vpcId": "",
```

Figure 18. Amazon ECS Task Definition

The agent reviews your environment and updates the CloudFormation template with configuration defined in the previous steps, results of application analysis (Figure 19).

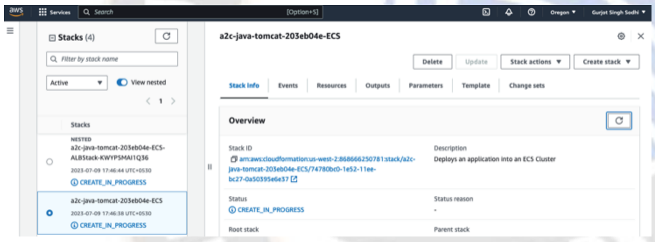


```
root@ip-10-0-0-89:~# cat /root/app2container/java-tomcat-203eb04e/EcsDeployment/ecs-master.yml
AWSTemplateFormatVersion: 2010-09-09
Description: Deploys an application into an ECS Cluster

Metadata:
  AWS::CloudFormation::Interface:
    - Label:
        default: App Service Parameters for the application being deployed, check ecs-lb-webapp.yml for usage
      Parameters:
        - DeploymentId
        - ContainerImage
        - RulePath
        - TaskCPU
        - TaskMemory
        - TaskCount
        - LogRetention
        - HealthCheckPath
        - HealthCheckCommand
        - OperatingSystemFamily
    - Label:
        default: Logging Parameters for the application being deployed, check ecs-lb-webapp.yml for usage
      Parameters:
        - TaskLogDriver
        - MultipleDests
        - SingleDestName
        - IncludePattern
        - LogGrpName
        - LogStrmName
        - AutoCrtGrp
        - FirehoseStream
        - KinesisStream
        - KinesisAppendNewline
        - FirelensName
        - FirelensImage
        - ConfigType
        - ConfigPath
        - UsingCloudwatchLogs
        - UsingFirehoseLogs
        - UsingKinesisLogs
```

Figure 19. Snippet of the Cloudformation template generated by the Agent

Your application container image is already existing in the ECR (Elastic Container Registry) at the final stage, and an ECS Task definition is produced and registered. For you to automatically deploy and configure all necessary resources in AWS, the agent generated the cloudformation template. The cloudformation template may be used to launch your application, as a last step.



Stacks
NESTED
a2c-java-tomcat-203eb04e-ECS-ALBStack-KWYPSMAI1Q36
2023-07-09 17:46:44 UTC+0530
CREATE_IN_PROGRESS
a2c-java-tomcat-203eb04e-ECS
2023-07-09 17:46:38 UTC+0530
CREATE_IN_PROGRESS

a2c-java-tomcat-203eb04e-ECS

Stack ID: arn:aws:cloudformation:us-west-2:868666250781:stack/a2c-java-tomcat-203eb04e-ECS/74780bc0-1e52-11ee-bc27-0a50395e6e37
Description: Deploys an application into an ECS Cluster
Status: CREATE_IN_PROGRESS
Status reason: -
Root stack
Parent stack

Figure 20. Deploying your Application in Cloud using the template generated by Agent

V. CONCLUSION

Microservice-based architectures on AWS cloud offer several advantages for application transplantation. Firstly, the microservice approach brings benefits such as better maintainability, flexibility, scalability, and efficiency. Secondly, the usage of containers in microservice architectures provides an ideal host for small and self-contained microservices. Containers help in packaging all the dependencies of a network function into a single unit and make it easier to manage and deploy multiple containers on a large multi-cloud infrastructure. Thirdly, independent deployment of microservices allows for auto-scaling and efficient handling of workload spikes. Lastly, adopting a microservice architecture on AWS cloud allows for seamless integration with other AWS services such as AWS Lambda for serverless computing and AWS Identity and Access Management for secure authentication and authorization. Overall, the combination of microservice-based architectures and AWS cloud provides a powerful solution for application transplantation. By leveraging microservice-based architectures on the AWS cloud, companies can achieve a high level of flexibility and scalability for their application transplantation efforts.

The microservice architecture, coupled with the capabilities of the AWS cloud, offers a robust and efficient solution for application transplantation. Transplanting the selected applications from the application server to the cloud also results in increased productivity and cost savings.

The proposed approach is powered by AWS App2Container cuts down the transplantation effort and time as it automates the assessment of the application server and induces containerization, and also generates the artifacts needed to deploy the container images (application of interest) to AWS This results in quick transplantation cycles. It automates the tooling on many targets in a secure manner.

The table below highlights the automation that we get during each phase of the transplantation, and compares it to the effort and skills that would have been needed if these were done manually.

TABLE II. ACCELERATING TRANSPLANTATION USING CONTAINERIZATION

Transplantation Activity	Without proposed approach		With proposed approach
	Steps involved	Required Skills	
Feature Identification/ Discovery and Assessment of legacy system	Manual / Custom tools	Strong understanding of legacy system	Automated
Dependency mapping	Manual / Custom tools	Strong understanding of legacy system	Automated
Docker file creation of the identified application of interest	Manual	Docker / Container	Automated
Building the Docker Image (of application of interest)	Manual	Docker / Container	Automated
Pushing Docker Image to Amazon Elastic Container Registry (ECR)	Manual	Docker / Container	Automated
Creating Deployment Manifest for hosting containers on Amazon ECS or Amazon EKS	AWS	Docker/ Amazon ECS/ Amazon EKS	Automated

It is important to keep in mind that every customer portfolio and application requirements are unique. Therefore, it's essential to validate and review any transplantation plans with business and application stakeholders. With the right planning, engagement, and implementation, you should have a smooth and rapid journey transplanting your legacy application to AWS cloud with AWS Containers.

REFERENCES

[1] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 373–384, Baltimore, MD, USA, July 2015.

[2] Eddy BP, Kraft NA, Gray J (2017) Impact of structural weighting on a latent Dirichlet allocation-based feature location technique. Wiley J Softw Evol Proc 30:1–25.

[3] Sodhi,G.S., Rattan,D. An Insight on Software Features Supporting Software Transplantation: A Systematic Review. Arch Computat Methods Eng 29, 275–312 (2022).

[4] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation computer systems, 25(6):599–616, 2009.

[5] Weronika Łabaj. Goodbye microservices, hello right-sized services. http://particular.net/blog/goodbye-microservices-hello-right-sized-services, 2015. Accessed On: 01-06-2023.

[6] Chris Richardson. Microservices: Decomposing applications for deployability and scalability. https://www.infoq.com/articles/microservices-intro, 2014. Accessed On: 01-06-2023.

[7] Martin Fowler. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, 2014. Accessed On: 01-06-2023.

[8] Johannes Thönes. Microservices. IEEE Software, 32(1):116–116, 2015, Sam Newman. Building Microservices. " O'Reilly Media, Inc.", 2015.

[9] Sam Newman. Building Microservices. " O'Reilly Media, Inc.", 2015.

[10] Eric Evans. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

[11] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham. Component identification method with coupling and cohesion. In Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific, pages 79–86. IEEE, 2001.

[12] Cesare S, Xiang Y (2012) Software similarity and classification. Springer Briefs in Computer Science, Berlin.

[13] Wenyong H, Jiquan L (1995) Software transplantation and under- standing tool—the study and realization of the VAX-C decompile system. In: National air intelligence center, Jisuanji Gongcheng, China, vol 18, pp 1–4.

[14] Poe J, Hughes C, Li T (2009) TransPlant: a parameterized methodology for generating transactional memory workloads. In: Bradley JT, Conrad JM, Field AJ, Harder U, Riley GF, Knottenbelt WJ (eds) 2009 IEEE international symposium on modeling, analysis & simulation of computer and telecommunication systems, London, pp 1–10. https://doi.org/10.1109/MASCOT.2009. 5366659

[15] Fu HT (2014) Elimination simulation of incompatibility of software transplantation on different platform. Trans Tech Publ Appl Mech Mater 687–691:2989–2992. https://doi.org/10.4028/www.scientific.net/AMM.687-691.2989

[16] Marginean A, Barr ET, Harman M, Jia Y (2015) Automated transplantation of call graph and layout features into Kate. In: Barros M, Labiche Y (eds) Springer symposium on search based software engineering (SBSE), Bergamo, Italy, pp 262–268. https://doi.org/10.1007/978-3-319-22183-0_21

[17] Dash SK, Ashokbhai VP, Sanmugasundaram R, Srinivasan D (2016) Transplantation of U-boot and Linux Kernel to OMAP-L138. In: Proceedings of IEEE international conference on microelectronics, computing and communications (MicroCom), National Institute of Technology, Durgapur, India 2016, pp 1–5. https://doi.org/10.1109/MicroCom.2016.7522407.

[18] Zhang T, Kim M (2017) Automated transplantation and differential testing for clones. In: Uchitel S, Orso A, Robillard M (eds) Proceedings of the 39th IEEE/ACM international conference on software engineering (ICSE), Buenos Aires, Argentina, pp 665–676. https://doi.org/10.1109/ICSE.2017.67.

[19] Petke J, Harman M, Langdon WB, Weimar W (2018) Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Trans Softw Eng 44:574–594. https://doi.org/10.1109/TSE.2017.2702606

[20] Liu L, Mao X (2018) A study on code transplantation technique based on program slicing. In: Hong Y H, Ke G T, He W(eds) Proceedings of international conference on transportation & logistics, information & communication, smart city (TLICSC), advances in intelligent systems research, Chengdu City, China, 2018, pp 294–298. https://doi.org/10.2991/tlicsc-18.2018.47

[21] Wang S, Mao X, Yu Y (2018) An initial step towards organ trans- plantation based on GitHub repository. IEEE Access 6:59268–59281. https://doi.org/10.1109/ACCESS.2018.2872669

[22] R.Ruiz, K. Park and V. Ganzert, "Apocalypse: The end of Antivirus", Kindle, 2015, pp. 1-134

[23] James Temperton, "Code 'transplant' could revolutionise programming", 2015. WIRED.co.uk. https://www.wired.co.uk/article/code-organ-transplant-software-myscalpel

[24] Dumitru H, Gibiec M, Hariri N, Huang JC, Mobster B, Herrera CC, Mirakhorli M (2011) On-demand feature recommendations derived from mining public product descriptions. In: Taylor RN, Gall H (eds) Proceedings of 33rd IEEE international confer- ence on software engineering (ICSE), Honolulu, Hawaii, USA, pp181–190. https://doi.org/10.1145/1985793.1985819

[25] McMillan C, Hariri N, Poshyvanyk D, Huang JC, Mobasher B (2012) Recommending source code for use in rapid software prototypes. In: Glinz M, Murphy G, Pezze M (eds) Proceedings of 34th international conference on software engineering (ICSE), Zürich, Switzerland, pp 848–858. https://doi.org/10.1109/ICSE. 2012.6227134

[26] Classen A, Heymans P, Schobbens PY (2008) What's in a feature: a requirements engineering perspective. In: Fiadeiro J, Inverardi P (eds) Proceedings of international conference on fundamental approaches to software engineering (FASE), Berlin, Heidelberg, pp 16–30. https://doi.org/10.1007/978-3-540-78743-3_2

[27] Kang K, Cohen S, Hess J, Novak W, Peterson S (1990) Feature- oriented domain analysis (FODA) feasibility study. Software Engineering Institute, Carnegie Mellon University, pp 1–161

[28] Goguen JA, Linde C (1993) Techniques for requirements elicitation. In: Proceedings of the IEEE international symposium on requirements engineering (ISRE), San Diego, CA, USA, pp 152–164. https://doi.org/10.1109/ISRE.1993.324822

[29] Marcus A, Malefic JI, Sergeyev A (2005) Recovery of traceability links between software documentation and source code. Int J Softw Eng Knowl 15:811–836. https://doi.org/10.1142/S0218 194005002543

[30] Beck K (2003) Test-driven development: by example. Addison- Wesley Professional, Boston

[31] Huang JC, Czauderna A, Gibiec M, Emenecker J (2010) A machine learning approach for tracing regulatory codes to product specific requirements. In: Kramer J, Bishop J (eds) Proceedings of ACM/IEEE 32nd international conference on software engineering (ICSE), ACM, New York, NY, USA, pp 155–164. https://doi.org/10.1145/1806799.1806825

[32] Martin L. Abbott, Michael T. Fisher, Art of Scalability, The: Scalable Web Architecture, Processes, and Organisations for the Modern Enterprise, 2015, Addison-Wesley Professional.

[33] Taibi, Davide & Lenarduzzi, Valentina & Pahl, Claus. (2018). Architectural Patterns for Microservices: A Systematic Mapping Study. 10.5220/0006798302210232.

[34] Michael Ayas, H., Leitner, P. & Hebig, R. An empirical study of the systemic and technical migration towards microservices. Empir Software Eng 28, 85 (2023). https://doi.org/10.1007/s10664-023-10308-9

[35] Taibi, D., Lenarduzzi, V., Pahl, C. (2019). Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study. In: Muñoz, V., Ferguson, D., Helfert, M., Pahl, C. (eds) Cloud Computing and Services Science. CLOSER 2018. Communications in Computer and Information Science, vol 1073. Springer, Cham. https://doi.org/10.1007/978-3-030-29193-8_7
[36] Razzaq, A., and Ghayyur, S. A. K., A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges, Comput. Appl. Eng. Educ. 2023; 31: 421– 451. https://doi.org/10.1002/cae.22586
[37] Muhammad Hamza. 2023. Transforming Monolithic Systems to a Microservices Architecture. SIGSOFT Softw. Eng. Notes 48, 1 (January 2023), 67–69. https://doi.org/10.1145/3573074.3573091
[38] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science, pages 137–146, 2016