Uniwersytet Łódzki

Wydział Matematyki i Informatyki

Sermest Eren Yazici

397309

Bachelor's thesis written in

Faculty of Mathematics and

Computer Science,

under the supervision of

dr. Wojciech Horzelski, UŁ

University of Lodz

Faculty of Mathematics and Computer Science

Sermest Eren Yazici

Systematic Troubleshooting of Open-Source Applications in Azure App Service: Challenges, Strategies, and Best Practices

Lodz, 2024

Abstract

As technology evolves, it brings new problems. Some of them may need a professional to resolve the problems. For professionals tasked with maintaining the integrity and functionality of complex systems, these challenges often necessitate a nuanced and systematic approach to problem-solving. This thesis endeavors to address this need by providing comprehensive guidance to future troubleshooters, particularly those operating within the realm of open-source applications deployed on Azure App Service and similar environments. Drawing upon a synthesis of established industry best practices, cutting-edge diagnostic strategies, and rigorous root cause analysis techniques, the research seeks to develop a structured methodology for efficient issue resolution. By delving into the intricacies of troubleshooting within the unique context of Azure App Service and open-source applications, this thesis aims to empower readers

with the knowledge and skills necessary to navigate and overcome the myriad challenges they may encounter in their professional endeavors. Through a combination of theoretical frameworks, practical case studies, and hands-on exercises, this work seeks not only to impart essential troubleshooting techniques but also to cultivate a deeper understanding of the underlying principles that govern system behavior and performance. By fostering a holistic approach to problem-solving, the thesis aspires to foster a new generation of professionals equipped to tackle the evolving complexities of modern technology with confidence and competence

Contents

1: Introduction

How were people hosting their applications before Cloud and how was the world before cloud computing?

Perhaps some organizations were using their own computer systems and physical components relying on traditional On-Premises infrastructure to have their computing needs[1]. Organizations had to buy and maintain their own servers, networks, storage, routers, etc. They were responsible for protecting and storing their data on their physical storage.

They had a site that housed data centers and served external users. In a data center, there were physical devices, databases, and network components. If an organization would like to improve the quality of their infrastructure, then they would need to buy and maintain the components.

To set up a data center, organizations would need to invest in a physical location and invest in purchasing physical components and network components.

It is important to remember that these physical locations require electricity and backup power solutions. Even after setting up a data center, purchasing and investing in components, we may still have failures. Identifying the cause of failures, particularly hardware failures, was often quite challenging for organizations.

Essentially, the traditional on-premises infrastructure model required organizations to make significant investments in physical hardware and resources. From purchasing and maintaining servers to ensuring a reliable power supply, operational complexities were many. Despite these efforts, inherent vulnerabilities, such as hardware failures and Power outages, posed significant risks to business continuity and data integrity.

1.1 Overview of Cloud Ecosystems

Cloud Ecosystems make everything more efficient and secure for the companies. Instead of individual organizations, the cloud providers now manage the Data Centers and physical infrastructure. Changing the way services are delivered and accessed.

Cloud providers operate Data Centers. These physical sites host the physical servers and network equipment necessary to support cloud services. In conclusion, Cloud computing reduces the need for hardware and physical site investments.

Importance of applications, downtimes, and resource usages. Cloud computing provides scalability to enable companies to scale their applications depending on usage and load. Instead of companies, cloud providers invest in fully maintained and highly secured infrastructure with data centers all around the world.

1.2 Key Components of Cloud Computing

In the realm of cloud computing, Platform as a Service (PaaS) and Software as a Service (SaaS) represent two fundamental models that cater to different needs of software development and deployment.[2]

Platform as a Service, which allows developers to have full control over their applications, is a developer tool and deployment environment. Similarly, Software as a

[2]https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-paas#:~:text=Platform%20as%20a%20service%20(PaaS,%2C%20cloud%2Denabled%20enterprise%20applications.

Service delivers software applications over the internet, such as email services and customer relationship management software, and is referred to as a service in the cloud. Key components include hardware, which is essential as it provides power supply, memory, storage, and CPU. These features work together to ensure the performance and security of the services provided.[3]

Virtualization of the software has a very important role. Because it's decomposing data storage and computing power from hardware.

Another question comes in here: what about deployment and deployment models?

There are different deployment models. Even though infrastructure resources are the same, they are allocated differently. If a pack of virtualized resources is used by multiple users outside an organization, that is a public cloud.

A Cloud environment that is dedicated to only specific users of an organization and accessed behind the firewall. This is the benefit of private cloud services with virtualized infrastructure.

[3]https://www.datacenterdynamics.com/en/news/google-to-build-new-uk-data-center-campus-in-hertfordshire/

1.2.1 Introduction to Open-Source Software

Open-source software shows computer applications belonging to those who published source code in public for use, modifications, and republishing.

This means that anyone can view, change, and enhance the application's source code as well as share it with others. Open-source software creates collaboration among developers and developer communities, enabling quality development.

The main characteristics of open-source software include free distribution, access to source code, derivative works, community collaboration.[4]

[4] https://opensource.com/resources/what-open-source
https://en.wikipedia.org/wiki/Open-source_software

Chapter 2: Literature Review

As we navigate to the world of cloud computing, one question looms large; What are the principal challenges and complexities that occurs from the traditional model of hosting web applications on physical infrastructure?

One of the foremost challenges of the traditional hosting model is significantly financial and operational burden of maintaining and upgrading physical infrastructure.

Scalability under traditional model represents significant concern. The ability to react quickly to changing demands "Scaling resources Up or Down"[5] poses inherent challenges within the physical infrastructure itself. As traditional data centers are limited by their own physical components; any adjustment in the capacity whether an increase or decrease requires physical changes to the infrastructure. Potentially leading to investing on additional servers for scaling up or dealing with underutilized resources. But it can also lead to operational inefficiencies, the latency in scaling operations potentially reduce the ability to respond to the high loads.

Considering these subsequent challenges posed by traditional models, it becomes important to explore more agile, cost-effective, and scalable alternatives. This leads to

[5] https://www.cloudways.com/blog/horizontal-vs-vertical-scaling/

pivotal question: How organizations transcend these limitations to achieve better flexibility, scalability and efficiently hosting their web applications?

We will dive into Azure App Service. As we pivot to these innovative solutions, one wonders; What makes Azure App Service a compelling alternative for organizations seeking to navigate from traditional infrastructure?

2.1. Introduction to Azure App Service

Talking about the App Service, a fully managed platform for building, deploying, and scaling web apps. It offers a rich set of capabilities to web developers for creating enterprise-ready web applications. With Azure App Service, developers can quickly build, deploy, and manage powerful websites and web apps using a fully managed platform that takes case of infrastructure maintenance, such as server provisioning, patching, load balancing and potential rollups.

App Service supports various of programming languages, including .NET, .NET Core, Java, Ruby, Node.JS, PHP and Python, allowing developers to work with their preferred language. It also integrates with Azure DevOps, GitHub and Bitbucket, enabling continuous integration and deployment workflows that streamline the development lifecycle.

One of the key features of Azure App Service is its ability to scale on demand. It can automatically adjust the number of virtual machine instances running the app to

accommodate varying amounts of traffic. Additionally, it provides high availability and ensuring that web apps are accessible to users with minimal downtimes.

App Service Also offers a range of services to enhance web app functionality, such as authentication services, custom domain support, SSL, TLS, certificate binding, and application insights for monitoring and analytics. It supports both Windows and Linux-based environments and provides a variety of service plans to meet different needs and budgets.

In summary, Azure App Service is a robust and versatile cloud platform that simplifies the process of web app development and deployment, allowing developers to focus on creating high-quality applications without worrying about the underlying infrastructure. Its scalability, support for multiple languages and integrations and a suite of additional services make it more attractive choices for a business and developers looking to leverage the power of the cloud.[6]

2.2. Core Features and Capabilities of Azure App Service

App Service has various hosting options depending on what organization needs and what organization wants to develop. Web Apps, API Apps, Mobile Apps and Function Apps. Alongside with those options, as well as various programming languages and their frameworks, Azure App Service also support docker and containerization.

[6] https://learn.microsoft.com/en-us/azure/app-service/overview

While organizations hosting their applications, docker and containerization feature brings another option. Deploying their application as Docker container which can be hosted in azure as Web Apps for containers. This approach offers the benefits of containerization such as improved scalability, consistency across environments, and streamlined CI/CD Pipelines. Before discussing the deployments, it's important to understand the fundamental concept of containers and images.

According to Docker, a container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computer environment to another. Containers are isolated from each other and the host system, ensuring that they are consistent across different environments. On the other hand, a container image lightweight, standalone, executable package of software that includes everything an application needs to run: Code, runtime, system tools, and settings. Container images become containers at runtime and in the case of Docker containers, images become containers when they run on Docker Engine[7]. Available for Linux and Windows environments, containerized software will always run regardless.

[7] Use containers to Build, Share and Run your applications.
https://www.docker.com/resources/what-container/ (Access Date: 14 April 24)

Docker containers that run on Docker Engine:

- **Standard:** Docker created the industry standard for containers, so they could be portable anywhere.
- **Lightweight:** Containers share the machine's such as OS system kernel and therefore do not require an OS per application. That allows organizations to driving higher server efficiencies and reducing server and licensing costs.
- **Secure:** Applications are safer in containers and Docker provides the strongest isolation capabilities in the industry.

With this understanding of containers and images, we can explore how they are utilized in a CI/CD pipeline to automate the deployment process. In Azure Web Apps, you can deploy your containerized applications by pushing your Docker images to a container registry such as Azure Container Registry or Docker Hub, and then pulling these images into your Azure Web App instances. What is an instance? Instance in Azure Web Apps is essentially a virtual server that managed by the Azure Platform. When developers deploy an application to Azure App Service it runs on one or more instances of the Azure Web App environment. Each instance operates independently with its own resources such as memory and processing power. The number of instances can be scaled manually or automatically depends on the scenario of the application and the load that application facing. Instances are important because they determine the scalability and redundancy of the application. If the application requires more resources to handle additional load, it can be scaled out by adding more instances. This known as horizontal scaling. Conversely, if the load decreases it can be scaled in by reducing the number of instances. Azure handles

load balancing across multiple instances to ensure that user requests are distributed evenly. In containerized deployments, each instance of an Azure Web App can run one or more containers based on the Docker images developers have created. This allows for a highly scalable and flexible architecture, as developers can deploy the same container image across multiple instances or use different images for different instances as needed. With the grasp of what instances are and how they support the scalability and availability of web applications in Azure Web Apps, how CI/CD pipelines are playing pivotal role in streamlining the deployment process?

CI/CD is a method to frequently deliver apps by introducing automation into the stages of application development. The main concepts attributed to Continuous Integration and Continuous Deployment and continuous delivery. CI/CD pipelines are designed to mitigate risks in software delivery and to enable more responsive deployment cycle.

1. **Continuous Integration (CI):** Developers merge their changes back to the main branch as often as possible. The changes are validated by creating a build and running automated tests against the build. This helps to identify any integration errors quickly.[8]
2. **Continuous Deployment/Delivery (CD):** Once the build with the changes passes all automated tests, it is deployed to a staging or production environment automatically. In the case of Azure Web Apps, this means that the new version of the application is deployed to the appropriate instances.[9]

Azure DevOps Services, GitHub Actions, Jenkins, and other CI/CD tools can be used to set up pipelines that automatically build, test, and deploy code every time there is a code change, based on the rules that developers define. This automation extends to containerized applications, where CI/CD pipeline can build Docker images and push them into a container registry like Azure Container Registry. From there, Azure Web Apps can pull the latest image and deploy new containers to its instances, ensuring that the application is always up to date with the latest changes. By integrating CI/CD pipelines with Azure Web Apps, developers can ensure that their applications are reliably released to the users with minimal manual intervention, and the entire process from code commit to production can be monitored and automated.[10]

[8] https://learn.microsoft.com/en-us/devops/develop/what-is-continuous-integration
[9] https://learn.microsoft.com/en-us/devops/deliver/what-is-continuous-delivery
[10] What is Azure Pipeline https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops Access Date 05.04.2024

2.3 Best Practices for Azure App Service

2.3.1. **Microservices Architecture:** In a microservices architecture, an application is composed of small, autonomous services that work together. Each service is self-contained and should implement a single business capability. Services communicate with each other using well-defined API's and protocols such as HTTP/REST, gRPC, or message queues. This architecture enables individual services to be developed, deployed, and scaled independently.

- **Benefits:** Services can be scaled horizontally, meaning the developers can add more instances of a service to handle increased load. Fault isolation is inherent, as issues in one service do not directly impact others, in microservices, if one environment facing an issue the others will not be impacted. Teams can develop, deploy, and scale their services independently, using the best technology stack for each service. Smaller codebases are easier to understand and maintain.
- **Troubleshooting Considerations:** When an issue arises, it can be isolated to a specific service, making it easier to diagnose and fix. Each service should have its own monitoring and logging in place, providing granular insight into the system's health. Implementing

distributed tracing can help track a request as it flows through the services, which is invaluable for troubleshooting complex issues.[11]

2.3.2. **Blue Green Deployment:** Blue-Green deployment is a release management strategy that reduces downtime and risk by running two identical production environments. The "Blue" environment represents the current live version of the application, while the "Green" environment is the staging area for new version. Once the new version on the green environment is fully tested and ready to go live, the traffic is switched over. This switch can be achieved through various methods, such as updating the DNS records, changing load balancer settings, or using Azure Traffic Manager.

- ***Benefits:*** By ensuring that one environment is always live, there is no downtime when deploying new versions. If any issues are detected after the switch, the developers can quickly revert to the blue environment. The new version can be gradually rolled out to users to minimize the impact of changes. The Green environment can be used for final testing in a production-like setting, which can uncover issues that might not be detected in development or test environments.

[11] Design Pattern For Microservices (Access Date 13.04.2024)
https://learn.microsoft.com/en-us/azure/architecture/microservices/design/patterns
Microservices on Azure (Access Date 13.04.2024)
https://azure.microsoft.com/en-us/solutions/microservice-applications#:~:text=What%20are%20microservices%3F,t%20break%20the%20whole%20app.

- ***Troubleshooting Considerations****:* It's crucial that the blue and green environments are as identical as possible to avoid "it works on my local machine" problems. Both environments should be monitored closely to detect any issues as early as possible. Special care must be taken with database schema changes, as they need to backward compatible to allow for rollbacks.

2.3.3. **Health Checks and Load Balancing:** Health checks are checks are a crucial component of any distributed system, including web applications hosted on Azure App Service. Health checks involve setting up a monitoring system that periodically sends requests (Often HTTP Get Requests) to an endpoint which is a health check URL to verify that an instance of web application is running correctly. These endpoints typically return a status code indicating the health of application: a 200 OK status indicates healthiness, while any other status indicates a problem Load balancer use these health checks to make intelligent decisions about traffic distribution. If an instance fails a health check, the load balancer stops sending traffic to that instance and reroutes it to healthy ones.

- ***Benefits:*** By routing traffic away from unhealthy instances, the system ensures that users are less likely to encounter errors or outages. The system can automatically respond to instance failures by removing them from the pool which helps maintain steady application performance under various conditions. Load balancing allows the

system to handle more traffic by distributing the load across multiple instances, preventing any single instance from becoming a bottleneck.

- ***Troubleshooting Considerations:*** Implementing detailed health checks that not only check if the application is running but also verify database connectivity, external service dependencies and other critical components. Logging and alerting that ensures health check failures are logged and alerts are set up to notify the appropriate team for immediate action. Auto scale can be used with Health check to auto scale and instance restarts to automatically recover from instance failures.

2.3.4. **Configuration Management:** Configuration Management in the context of Azure Web Apps involves the practice of separating configuration settings from the application code. This is typically achieved by using environment variables, configuration files that are not part of the code repository or services like Azure Key Vault for storing secrets. Externalizing configuration settings means that the developers can store and manage settings such as database connection strings, API keys, feature flags, and other environment-specific values separately from the application codebase. In Azure Web Apps, the developers can manage these settings directly through the Azure portal, using Azure CLI, or Azure PowerShell.

- The developers can change configuration settings without modifying the code, which is particularly useful for settings that vary between environments (development, staging, production).
- Sensitive information like connection strings and API keys can be kept out of the codebase, reducing the risk of exposing them in source control. Developers can focus on code, while operations teams can manage configurations independently. It is easier to manage and maintain settings when they are centralized and not scattered throughout the code.
- Configuration settings can be easily scaled or modified to accommodate different instances or environments.

- ***Troubleshooting:*** When issues arise, having externalized configurations settings simplifies the troubleshooting process: The developers can quickly adjust configuration settings in response to issues such as performance bottlenecks or service outages without the need for code changes or redeployments. Since configuration changes are isolated from the code, the developers can track and audit them independently, which helps in identifying what change might have caused an issue. If a configuration change does not resolve the issue or creates new one, it's easier to roll back to a previous configuration state and developers can test the impact of configuration changes in

isolation from code changes, which helps in identifying root cause of issues.

2.3.5 Azure DevOps and Local Git Deployments: A Beginner's Guide

DevOps pipelines are automated processes that help teams integrate new code changes into their projects and ensure that these changes are don't break the existing functionality. They are crucial part of the DevOps practice, which aims to unify software development (DEV) and software operations (OPS).

What does a Pipeline Consist Of?

A pipeline is made up of several components:

- **Stages:** These are logical boundaries in the pipeline where the developers can group jobs. For example, the developers might have a build stage, a test stage and a deployment stage.
- **Jobs:** A job is a sequence of steps that run sequentially on the same agent. The developers might have separate jobs for building the code, running tests and deploying the application.
- **Steps:** These are the smallest units of work, such as executing a script, compiling code or running tests. Each step is run in order within a job.

` ***How to Trigger a Pipeline?***

Pipelines can be triggered in several ways:

- **Manually:** A user can start a pipeline run through the Azure DevOps user interface.
- **Schedule:** Pipelines can be configured to run at a specific time, such as nightly builds.
- **Source Control Events:** Most common trigger. Whenever a developer commits new code or merges a pull request into the main branch, it can automatically trigger the pipeline to run.[12]

Build Pipelines vs Release Pipelines:

- **Build Pipeline:** These compile the code, run tests and create artifacts[13]. The main goal is to validate that the new code changes integrate well with the existing codebase and that they don't introduce any errors.[14]
- **Release Pipelines**: Once the build pipeline has created an artifact, the release pipeline takes over. It handles the deployment of these artifacts to different environments, such as development, staging, and production.

[12] https://learn.microsoft.com/en-us/azure/automation/source-control-integration Access Date 12.04.2024
[13] Packages that can be deployed.
[14] Managing Build Pipelines https://docs.oracle.com/en-us/iaas/Content/devops/using/managing_build_pipelines.htm#:~:text=A%20build%20pipeline%20contains%20the,action%20in%20the%20build%20pipeline. (Access Date 16.04)

The release pipeline can also manage the configuration and provisioning of the infrastructure needed for the application to run.[15]

Local Git:

Local Git revers to a Git repository that resides on local machine. Git is a version control system that allows developers to track changes in codebase, collaborate with others, and revert to previous states of the project if necessary. When the developers use Local Git with Azure App Service, they are essentially using Git as a means to deploy their code directly from their local machine to the cloud. But what is Git? Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It allows multiple developers to work on the same codebase without interfering with each other.

[15] Release Pipeline https://www.professional-devops.com/release-pipeline.html#:~:text=A%20release%20pipeline%20is%20a,%2C%20testing%2C%20and%20collecting%20results. (Access Date 16.04)

What is Local Git Repository?

A local Git repository is a directory on the computer where Git tracks the changes of the code. It's "local" because it resides on the machine, as opposed to a remote repository which is hosted on a server (like GitHub, Bitbucket, or Azure Repos).

How Does Local Git Works with Azure App Service?

Azure App Service supports deployment from various sources, one of which is a local Git repository. The developers can push their code changes from their local repository to Azure, and it will automatically deploy the new code to the web app. Setting Up Local Git For Azure Deployment: The developers start by creating a local Git repository in their project's directory using the **git init** command. After making changes to their code, they use **git add** to stage changes and **git commit** to save them to their local repository. Azure App Service provides a Git URL that the developers can add to their local repository as a remote. This is done using the **git remote add** command. Once they have committed their changes and set up the remote, they can push their code to Azure using the **git push** command. Azure will then receive the code and start the deployment process.

With Local Git, developers can push updates to Azure App Service anytime they have changes. Each push can trigger a deployment, effectively giving them continuous deployment capabilities. This means that everytime they push their code changes, Azure App Service will automatically deploy the new version of the application.

When developers push their code to Azure App Service, using Local Git, Azure Receives the code and triggers an automated deployment process. This process includes

restoring dependencies, compiling the code if necessary, and starting the new version of the application. Azure handles these steps according to the configuration the developers set for their application.

Chapter 3: Methodology

Troubleshooting is a critical aspect of maintaining the operational health and performance of any technological system. In the domain of cloud computing, particularly with complex platforms like Azure, the diversity and intricacy of the infrastructure demand not just a robust approach to troubleshooting, but a comprehensive, systematic one. This necessity arises from the need to manage and rectify issues across an extensive array of services that Azure offers. The general troubleshooting methodology for Azure is far from a static checklist. It is a dynamic, adaptive framework that evolves alongside the technology it supports, designed to address the unique challenges that emerge. At its core, this methodology acknowledges that effective troubleshooting is both an art and a science—it demands a balanced application of technical expertise, logical reasoning, creativity, and intuition. Our approach is structured yet flexible, tailored to address the unpredictable nature of technical issues while providing clear, actionable guidance. The troubleshooting process is inherently iterative, often requiring multiple evaluations and tactics to pinpoint and resolve problems. It emphasizes an evidence-based strategy, utilizing data and empirical

results to steer decisions, ensuring a thorough and exhaustive exploration of potential solutions. This methodology is crafted to leave no stone unturned, guaranteeing a comprehensive search for resolution and fostering continual improvement in handling the complexities of Azure's vast ecosystem.

3.1 Identifying and Diagnosing Issues

In the context of troubleshooting within Azure or any complex system, it is crucial to have clear understanding of certain terms that are often used interchangeably but have distinct meanings. These terms are the building blocks of effective communication and problem-solving.

- ***Issue:*** An issue is a broad term that refers to any deviation from the expected behaviour of a system. It encompasses any situation where the system performance, functionality, or reliability is compromised. Issues can range from minor inconvenience to major disruptions.
- ***Fault:*** A fault is a specific defect within a system that has the potential to cause an error. It is underlying problem that exists within the system's hardware, software, configuration, or process. A fault might not always lead to an immediate failure but represents a flaw that could trigger an issue under certain conditions.
- ***Failure:*** Failure is the manifest of a fault. It occurs when a system component or function does not perform as intended due to a fault. Failures can be partial, where some functionality is lost, or total, where the system becomes completely inoperative.

- ***Symptom:*** A symptom is an observable indication that a failure has occurred. It is the evidence of the issue that users or monitoring tools can detect. Symptoms can be misleading, as they often resemble the effects of other unrelated faults, which is why they must be carefully analysed to trace back to the actual fault.

3.1.1 The Importance of Systematic Approach

The significance of a systematic approach in troubleshooting cannot be overstressed, particularly in the context of complex systems like those encountered in Azure. A systematic approach to problem-solving is a disciplined methodology that involves following a consistent and logical sequence of steps to diagnose and resolve issues.

One of the primary benefits of a systematic approach is the efficiency it brings to the troubleshooting process. When faced with an issue, there may be a multitude of potential causes. A systematic approach helps to identify quickly and accurately the most likely causes and rule out the less likely ones, thereby saving time.

This efficiency is particularly important in a production environment where downtime can be costly. By following a structured methodology, developers or engineers can minimize the impact of issues on end-users and the business operations that depend on Azure services.

In addition to efficiency, a systematic approach ensures that the analysis is thorough and comprehensive. It requires that all aspects of the system and its environment are considered. This includes not just the software and hardware but also the network infrastructure, dependencies, configurations, security settings, and even external factors such as third-party services.

By being methodical, developers and engineers are less likely to overlook a potential cause of the issue, which otherwise lead to a misdiagnosis and ineffective solutions. Troubleshooting in a systematic way also ensures repeatability and consistency. When developers follow a set process, they create a record of their steps that others can follow.

This is invaluable for knowledge sharing and for ensuring that the same high standards are maintained across different shifts, teams, and even geographical locations[16]. Consistency in the approach means that when similar issues arise, they can be addressed with the same level of expertise and efficiency, regardless of who is doing the troubleshooting.

Approaching systematically contributes to knowledge building within the organization. As issues are encountered and resolved, the knowledge gained can be documented and used to train new engineers. This creates a knowledge base that can be referenced in the future, which not only speeds up the resolution of new issues but also helps in preventing the recurrence of past issues.

[16] In term of support, if the issue is not resolved in the shift time, the current engineer may take troubleshooting notes for the next available engineer.

By following a systematic approach, engineers can mitigate risks associated with changes and interventions. Complex systems often have interdependencies, and a change in one can have unintended consequences in another. A methodical approach includes steps for risk assessment, change management, and rollback procedures in case the proposed solution does not work as expected or exacerbates the issue.

By breaking down the problem and examining each component and its interactions with the rest of the system, engineers can isolate where the issue is occurring. Once isolated, a deeper analysis can be conducted to determine why the issue occurred, which is critical for implementing a permanent fix rather than a temporary workaround.

3.1.2 Problem Identification

Problem identification is a critical stage in the systematic approach to troubleshooting Azure Web Apps or any complex system. It involves recognizing and categorizing symptoms as they appear, which sets the stage for effective diagnosis and resolution. The process of problem identification requires maintaining an analytical mindset and adhering to a systematic approach to ensure that the true nature of the issue is understood and addressed.

- ***Recognizing Symptoms:*** The first step in problem identification is to recognize the symptoms that indicate an issue. Symptoms can manifest in various ways,

such as error messages, performance degradation, unexpected behaviour, or complete service outages. It's essential to observe these symptoms carefully and document them accurately. For Azure Web Apps, symptoms could include:

1. Http error codes (e.g. 500 Internal Server Error, 503 Server Unavailable)
2. Slow page load times or timeouts.
3. Application exceptions or crashes.
4. Inability to deploy or update the web app.
5. Connectivity issues with databases or other services.
6. Security breaches or unusual access patterns.

- **Categorizing Symptoms:** Once symptoms are recognized, the next step could be categorizing them. Categorization involves classifying the symptoms based on their characteristics, which can provide insights into the potential causes. Symptoms can be categorized by:
    1. Type (e g., performance-related, connectivity-related, security-related)
    2. Severity (e g., critical, major, minor)
    3. Frequency (e.g., intermittent, continuous)
    4. Scope (e.g., affecting all users, specific to a region, limited to certain functionalities)
- ***Maintaining an Analytical Mindset:*** An analytical mindset is essential during problem identification. It involves being objective, critical, and logical when assessing symptoms. If engineers would avoid jumping to conclusions or making assumptions without evidence, it would be ideal. They should:

1. **Gather data:** Collect logs, metrics, and user reports that pertain to the symptoms.
2. **Analyse Patterns:** Look for patterns in the data that may indicate commonalities or triggers for the issue.
3. **Ask Questions:** Formulate questions that can help narrow down the cause, such as “When did the issue start?” or “What changes were made recently?”

- ***Systematic Approach in Problem Identification****:* A systematic approach in problem identification involves following a structured process to ensure that no aspect is overlooked. This approach typically may include:
    1. **Documentation:** Keeping a detailed record of the symptoms, data collected, and action taken.
    2. **Communication:** Informing stakeholders about the issue and ongoing efforts to resolve it
    3. **Replication:** Attempting to replicate the issue in a controlled environment.
    4. Elimination: Using a process of elimination to rule out potential causes that do not match the symptoms.
    5. **Escalation:** Knowing when to escalate the issue to more experienced team members or to seek assistance from senior developers.
- **Importance of Systematic** Approach and Analytical Mindset: The importance of a systematic approach and an analytical mindset in problem identification cannot be overstated. These elements are critical for several reasons:

1. They prevent hasty decisions that could lead to incorrect diagnoses or exacerbates the issue.
2. They help in identifying not just the symptoms but also the underlying causes, which is essential for a long-term resolution.
3. They ensure that the troubleshooting process is transparent and can be reviewed or audited if necessary.
4. They build confidence among users and stakeholders that the issue is being handled professionally and with due diligence.

3.1.3 Diagnostic Strategies: Diagnostic strategies can be essential for effectively troubleshooting issues with web apps. By replicating problems, isolating the affected systems components, and employing root cause analysis techniques, engineers can identify and resolve the underlying causes of issues. Here we will delve into these strategies in depth.

1. **Understand the symptoms:** Gather as much as information as possible about the issue from logs, user reports and monitoring tools.
2. **Create a Controlled Environment:** Setting up a non-productive environment that closely mirrors the production setup where the issue occurs.
3. **Mimic the Conditions**: Try to replicate the exact conditions under which the problem was reported, including user actions, data inputs, network conditions, and configurations.
4. **Automate Replication:** If the issue is intermittent or complex, consider scripting the actions that lead to the problem to facilita repeated testing.

3.1.4 Isolating the Issue: Once the problem is replicated, the next step is to isolate the part of the system where the issue is occurring. Isolation techniques can include:

1. **Component Testing:** Breaking down the application into its individual components (e.g., Front-end, back-end, database) and test each one separately to identify the faulty component.
2. **Change Analysis:** Reviewing the recent changes to the application or environment to determine if the issue correlates with a specific update or configuration change,
3. **Dependency Checking:** Verify the status and health of all external dependencies, such as databases, APIs, or third-party services, to rule them out as the cause of the issue.
4. **Log Analytics**: Examine application and server logs for errors or anomalies that can pinpoint where the problem lies.

5. **Performance Metrics:** Analyse performance metrics to identify resource bottlenecks or failures.

3.1.5 Root Cause Analysis Techniques: Root cause analysis (RCA) is a method of problem-solving used to identify the underlying reasons for faults or problems. One RCA technique is the "5 Whys", which involves asking "why" multiple times until the root cause is revealed. Here is how to apply it[17]:

1. **State the problem:** Clearly define the problem that you are trying to solve.
2. **Ask the first Why:** Ask why the problem is occurring and identify a cause.
3. **Dig Deeper:** For each answer provided, ask "why" again to uncover the next layer of causation.
4. **Continue the Process:** Repeat this questioning process until you reach the root cause, which may occur by the fifth "why".
5. **Verify the Root Cause:** Ensure that addressing the identified root cause would prevent the problem from recurring.

[17] https://en.wikipedia.org/wiki/Five_whys Access Date 20.04. (Access Date 20.04.2024)

3.2 Decision Making in Troubleshooting

Perhaps one of the critical skills that involves determining the most appropriate course of action to address an issue based on its impact and urgency. The ability to prioritize troubleshooting actions effectively can mean the difference between a swift resolution and prolonged downtime. When faced with a problem, especially in the context of Azure Web Apps, engineers could weigh the benefits of implementing a quick fix against the time and resources required for a more permanent solution.

3.2.1 Prioritized Based on Impact and Urgency

The impact of an issue refers to the extent to which it affects users, business operations, and the overall system. Urgency relates to how quickly the issue needs to be resolved to prevent further consequences. To prioritize effectively, engineers could consider following:

- **Severity of the Issue:** How critical is the problem? Does it cause a complete outage, data loss or security breach. Or is it a minor glitch affecting a small number of users?
- **Scope of the Issue:** Is the problem widespread, affecting all users and services, or is it localized to a specific component or user segment?
- **Business Priorities:** What are the business hours or peak usage times? Are there any important events or deadlines that would increase the urgency of resolution?

- **Resource Availability:** What resources (personnel, tools, etc.) are available to address the issue, and how might this affect the time to resolution?

By assessing these factors, engineers can categorize issues into a matrix of high/low impact and high/low urgency to determine the appropriate response.

3.2.2 Quick Fix vs Permanent Solution

Once the issue has been prioritized, the next decision can be whether to apply a quick fix or invest time in a more permanent solution. This decision could depend on different factors:

- **Quick Fix**: A quick fix, also known as a workaround, is a temporary solution that restores service or functionality in the short term. Quick fixes can be beneficial when:
  - The issue has a high impact and high urgency, and immediate action is required to mitigate negative effects.
  - A permanent solution would take too long to implement, and the cost of prolonged downtime is unacceptable.
  - The root cause is not immediately clear, and more time is needed for a thorough investigation.
  - The quick fix has minimal risk of causing additional issues or side effects.

- **Permanent Solution:** A permanent solution addresses the root cause of the problem and aims to prevent it recurrence. Permanent solution can be preferable when:
    - The issue has a lower urgency, allowing time for a proper fix without severely affecting users or business operations.
    - The root cause is known, and a reliable fix can be implemented without causing further disruption.
    - The quick fix would only delay the inevitable need for a more robust solution, potentially at a higher cost.
    - The issue is recurrent, and the cumulative impact of repeated quick fixes is greater than implementing a permanent solution.
- **Decision-Making Process:** The decision-making process in troubleshooting can involve the following steps:
    - **Assess the Situation:** Quickly gather all relevant information about the issue to understand it's impact and urgency.
    - **Consider the Option:** Evaluate the feasibility and implications of both quick fixes and permanent solutions.
    - **Risk analysis:** Analyse the potential risk associated with each option, considering factors such as system stability and future maintainability.
    - **Consult and Communicate:** Troubleshooting is likely to solving geometry. Share perspectives and involve the team and communicate with stakeholders to gain insights and align on the chosen course of action.

- **Implement and monitor:** Once a decision is made, implement the chosen fix and closely monitor its effectiveness.
- **Review and Learn:** After the issue is resolved, review the decision-making process and outcomes to learn from the experience and improve future troubleshooting efforts.

To maintain a high level of service and system reliability, it is essential for engineers and support teams to commit to continuous improving their diagnostic skills and methodologies. This involves staying updated with latest Cloud features and tools, participating in training and development opportunities, and fostering a culture of collaboration and knowledge sharing. By doing so, teams can enhance their ability to respond effectively to issues, reduce resolution times, and prevent future problems. Remember, troubleshooting is not just about fixing problems- it is about learning from them to build more resilient and robust systems.

Chapter 4: Findings and Analysis

4.1 Common Challenges Encountered

***Web App Experiencing High CPU*:** User reports that they are seeing high CPU usage on an App Service Plan, App Service, or a specific instance.

During the initial scoping of the problem, we want to answer the following questions. These can be answered by asking the user or by utilizing the tools.

1. Where is the CPU occurring? Is it affecting the entire App Service Plan, one or more App Services in the plan, all instances, or a subset of the instances?
2. Which metrics is the user using to assess that the CPU is high?
3. What is the CPU percentage that we can observe?
4. How many sites (including slots) are running in the App Service Plan and which web jobs run in each App Service?
5. How does the current (problematic) CPU Percentage compare to normal behaviour?
6. What is the size of the instance? (Small, Medium, Large)

We can approach with a way of determining or validating which App Service Plan Experiences High CPU. The App Service plan is the set of instances on which multiple applications can run. All App Services(application) will contribute to the overall App Service Plan CPU Utilization to different degrees.

We can start determining which instances are experiencing the high CPU and to what degree. Then isolate which App Service contributes the most to the high CPU.

In some cases, there can be multiple high contributors to the overall plan-level utilization.

In the affected App Service application(s), isolate which process is consuming the most CPU. This will usually be the application process, a web job process or a platform/system component. Determine if the high CPU utilization is mostly caused by the application, or if it is caused by a platform component. In most cases it will be the application or web job.

If necessary, try to isolate which specific area of user's code is consuming too much CPU. This type of analysis is advanced and is not usually necessary.

App Service Plan CPU Usage: After validating which ASP is experiencing high CPU. All applications on a given plan contribute to the overall usage. This step is typically trivial if we know the name of at least one of the applications in the plan.

From App Service Plan Metrics, we can analyse by reviewing the plan CPU and in the diagnose and solve problems we can see the instance CPU usages. We may see different patterns such as:

- All instances have a similar CPU usage over time, and all of them reach high CPU during a particular time.
- Different instances show high CPU at different times.
- A small number (or one) instances showing a sudden spike, then usage drops down.
- Some instances experience a jump to a high level which never comes down.

- Some combination of these patterns.

Code Level CPU Usage: Analysis beyond some points typically focuses on what specifically in the application consumes extra CPU. Moving in this direction is typically much more difficult and time intensive than the steps taken up to this point, A decision to further analyse must be carefully weighed against other options. In many cases, mitigation or factor elimination is a much faster and efficient way forward.

Mitigation: Mitigating the issue perhaps much more important than a detailed analysis. Mitigation tools can be used to resolve the issue but also as a method to formulate or test hypotheses about what is causing the issue.

Scaling is up or out can mitigate or resolve a larger number of CPU usage related issues. Scaling up (Small to Medium, or Medium to Large) could be a first step as scaling up is typically a better mitigation for memory issues as well, and there is often an overlap between CPU and memory usage issues. Scaling up each tier doubles the number of cores/CPU per machine, so it has a big effect on increasing processing power of each instance.

Web App Experiencing High Memory:

The first and foremost question to understand is what "memory" user is talking about here and where the user is noticing high memory. These concepts are important to understand.

- **Ram:** Often referred to as physical memory. This is fixed on an instance.

- **Private Bytes:** Refer to the amount of memory that the process executable has asked for – not necessarily the amount is using. They are "private" because they usually exclude memory-mapped files (i.e shared DLLs). But there's the catch – they don't necessarily exclude memory allocated by those files. There is no way to tell whether a change in private bytes was due to the executable itself, or due to a linked library. Private bytes are also not exclusively physical memory; they can be paged to disk or in the standby page list.
- **Working Set**: Refers to the total physical memory (RAM) used by the process. However, unlike private bytes, this also includes memory-mapped files and various other resources, so it's an even less accurate measurement than the private bytes. This is the same value that gets reported in Task Manager's "Mem Usage". Memory in the Working Set is "physical" in the sense that it can be addressed without a page fault.
- **Virtual Bytes:** They are total virtual address space occupied by the entire process. This is like the working set, in the sense that it includes memory-mapped files. But it also includes data in the standby list and data has already been paged out.[18]

Scenario: User complains that Web App is consuming High Memory or the instance serving the Web App has less available memory left.

[18] https://stackoverflow.com/questions/1984186/what-is-private-bytes-virtual-bytes-working-set

Scoping:

- Where the user seeing High Memory? Is he noticing low physical memory or low available memory on the instance or is he noticing High Private bytes for his worker process?
- How much memory consumption is he seeing?
- How many sites/ slots/ web jobs are running in the App Service plan?
- Why it is important for the user even if memory is high? Is he seeing slow performance due to high memory and has he isolated it in some way?
- Is the user seeing some recycle events due to auto healing? If yes, we need t to capture the exact event and understand what exactly is logged in that event.

We can check the app's performance from the diagnose and solve problems Web App Down, and switch to the requests and performance view separately and confirm if there are more failure requests or the slower response during the high memory time. If the memory usage was spiked in a short time and the memory usage was dropped down automatically without restarting the site. And there is no obvious application's performance was impacted which means the application was busy with processing some data and it did need that kind of machine resource during that time. We can utilize Memory Drill Down tool to check the memory usage at the process level.

HTTP 500 Errors*:* User is reporting that the application is down, and requests return a status code of 500. User is reporting that the application is crashed or threw an exception but has not specifically stated that the site returns a status code of 500.

As a rule, a 500-status code indicates that a component running in the web server process had an unhandled exception and the web server had to provide the exception handling on behalf of the that component. As a result, the web server is reporting that the request was unable to complete correctly, and the web server reports that as a status of 500. In most cases this unhandled exception originated in the chosen application framework (such as Asp.Net, PHP, Java or etc.) and /or the application code written by the user. However, this exception could have been caused by an underlying platform issue, such as a disk, storage issue, network issue, etc.

Since the exception typically originates in the application code, customer participation is often necessary to uncover the source of the unhandled exception.

HTTP 502 Errors: User is reporting that the application is down and requests return a status code of 502.

The HTTP 502 Bad Gateway server error response code indicates that the server, while acting as a gateway or proxy, received an invalid response from the upstream server. In other words, the first server (or process) receives the request and forwards it to a second server (or process). The first server does not receive a response from the second server and reports a 502 error.

There could be many reasons why the upstream server does not respond. An app running in the upstream server may have an unhandled exception or could not respond in time. Since the server acting as gateway or proxy does not know te reason, it will report that as a status of 502. The cause may originate in the chosen application framework (such as Asp.Net, PHP, Java etc.)

If status code of 502 is ONLY reported in Front-end, suggested way to check status code of the corresponded request in worker.

- There is a 4-minute timeout on the TCP level between the Front End and Workers. If the value of time taken is about 4-minute, there may be resource exhaustion such as Memory or CPU.
- If status code is 200 and time taken is very small (less than 4 minutes), it is very possible that the 502 error is "502.3 error with wind32 code 1229." It usually means the client disconnected the connection before the server sent back the response[19]. The disconnection may be initiated by the user or any devices between the user and Azure.

If there is no corresponded request recorded within instance:

- There can be issues within platform that are typically associated with these events: storage issues, out of memory errors, outbound socket exhaustion. If these events correlate with the time of the 502 errors and the 502 errors do

[19] https://stackoverflow.com/questions/30706400/an-operation-was-attempted-on-a-nonexistent-network-connection-error-code-1229

not occur outside of the time window of these types of failures, it is highly likely that they are the causes of these 502 status code.

HTTP 503 Errors: User is reporting that the application is down, and requests return a status code of 503.

The HTTP 503 Service Unavailable server error response code indicates that the server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded[20]. In most cases this issue is originated in the chosen application framework such as (Asp.Net, PHP, Java etc.). Since the cause typically originates in the application code, user participation is often necessary to uncover the cause. Platform logging captures many common scenarios, such as a storage failover, but we do not have exhaustive logging for every potential cause.[21]

Docker Configuration/Code: Sometimes we may see issues where the container can't start, could be due to configuration, code or even platform issues. On this part we are going to troubleshoot the coding and configuration issues.

First, we need to check user's **linux_fx_version** which defined in the Docker file such as

For Azure provided containers we will see "**linux_fx_version**": "**NODE|16-lts**", or "**PHP|8.2**", etc.

[20] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503

For Web App for Container/Custom Image we will see the **linux_fx_version:** "**DOCKER\<dockerhub/ACR Account name>/image:tag**"

When troubleshooting an application that is failing to start – and combing through App Service logging, this is one of the first maybe more common messages we may see. App Service on Linux (for both Azure provided and custom Docker Images) requires that the application container send back an HTTP response to the internal platform ping that is sent – which dictates that the container is started and in general, able to return a response. This platform ping is sent over HTTP to the root of the site.

This message indicates that the application did not send back an HTTP response to the platform ping in the predetermined time. By default, this time is 230 seconds before it times out- in which case, the message is **container did not respond to HTTP pings on port, failing site start** will be returned and the site will attempt to start the container again after some time.[22]

Scenarios: Incorrect Port Values

For Azure provided images, the docker container used for the application stack that the user deploying to has a certain default exposed port. If the container is listening on different port on the application server versus what's actually exposed when the container is ran, then the container will time out, as the request is not being forwarded to the application port as it's not properly listening on the exposed port. We can change this with the **PORT** App Setting.

[22] https://azureossd.github.io/2023/04/17/Troubleshooting-Container-didnt-respond-to-HTTP-pings-failing-to-start-site//

For Custom images we can use **WEBSITES_PORT**

***Scenarios*:** Binding to localhost

Applications deployed to App Service Linux should not be attempting to bind to localhost or 127.0.0.1.[23] This listening address is normally passed to what the Web/Application server should be listening on[24]. Any external requests trying to be made to the container will never get a response in this case- and in this case, platform pings will also not get a response, timing out the container. Applications should be listening on 0.0.0.0 instead. [25]

Applications that have a very long startup time before an HTTP response is sent back may encounter this error described.

Some scenarios are:

- Data transformation
- Calling to external dependencies to grab data needed at startup
- Installation of packages or other runtime configuration
- Etc.

If any of this causes the container to not start by 230 seconds, the container will time out. This can be extended with the App Settings

[23] https://learn.microsoft.com/en-us/answers/questions/1532160/flask-web-app-runs-locally-but-shows-application-e

[24] https://azureossd.github.io/2023/04/17/Troubleshooting-Container-didnt-respond-to-HTTP-pings-failing-to-start-site/

[25] https://azureossd.github.io/2023/04/17/Troubleshooting-Container-didnt-respond-to-HTTP-pings-failing-to-start-site/

WEBSITES_CONTAINER_START_TIME_LIMIT to a maximum of 1800 seconds. Most times, container timeouts are a symptom of issues not related to long startup times.

Route is not configured to return an http response[26]. In some scenarios, an application may actually not be developed to return an http response on the root (/) path. Although rare, it is best to ensure an actual http status code is returned. In some cases, fallback/catch-all logic is introduced for non-matchin routes, on application like API's – ensure that these catch-alls actually have a logic to return some type of http response.

Missing Environment Variables: Some applications depend on environment variables for certain environments. On App Service, these are added via the **App Settings** functionality, which is essentially environment variables. If the application requires these, but is not added – depending on the application logic, it can cause the container to time out.[27]

Incorrect Startup Commands: Applications that use certain startup commands depending on their environment may encounter this behaviour. For instance, depending on the environment, a command meant to run against development resources rather than production-based ones may be used- which may bind to **localhost**, a different port, or a local databases. Ensure the startup commands being used is correct.

[26]

[27] https://azureossd.github.io/2023/04/17/Troubleshooting-Container-didnt-respond-to-HTTP-pings-failing-to-start-site/

Installing Packages on Startup: This cause is slightly different than the Long application startup routines and logic. As this is more specifically talking about the startup command option for Azure provided images.

This applies to installing either language/stack specific packages or Linux specific ones. Depending on the package size – and number of packages – this may introduce extended startup time in these scenarios, as downloading these packages will need to go out over the internet.

Conclusion

The systematic troubleshooting of applications is a critical process that demands a comprehensive and adaptable methodology. By following the structured approach outlined, developers and engineers can effectively navigate the complexities of the Azure ecosystem and resolve issues efficiently.

The book highlights the importance of understanding Azure Devops pipelines, which automate the build, test, and deployment processes. Mastering concepts like stages, jobs, steps, triggers, build pipelines, and release pipelines lays the foundation for streamlining application delivery and troubleshooting deployment-related issues.

It delves into the intricacies of deploying web applications to Azure App Service using Local Git. Grasping the deployment workflow, from pushing code changes to Azure's automated processes like dependency restoration and code complication, equips developers with the knowledge to troubleshoot deployment failures and application startup issues.

The crux lies in presenting a robust methodology for troubleshooting Azure web apps. It emphasizes the significance of a systematic approach, starting with precise problem identification through symptom recognition and an analytical mindset. Diagnostic strategies like replicating issues in controlled environments, isolating affected components, and employing root cause analysis, techniques like the “5 Whys” enable pinpointing the underlying causes accuracy.

The importance of decision-making skills in troubleshooting is also highlighted. Prioritizing actions based on impact, urgency and weighing quick fixes versus permanent solutions is crucial for mitigating risks and ensuring business continuity. Assessing factors like severity, scope, business priorities, and resource availability guides this prioritization process.

Common challenges faced when troubleshooting Azure web apps are addressed, such as high memory usage, HTTP errors like 500 (unhandled exceptions), 502(Bad Gateway) and 503 (Service Unavailable). Insights into troubleshooting steps, including analysing metrics, logs, replicating conditions, isolating components, and conducting root cause analysis are provided.

Challenges like extended startup times due to package installations and platform issues like storage, memory, or socket exhaustion are also covered, emphasizing the need for careful investigation and resolution.

Throughout the troubleshooting process, documenting steps, communicating with stakeholders, and building a knowledge base from resolved issues contribute to

continuous improvement and the development of a robust troubleshooting culture within organizations.

REFERENCES

https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-paas

https://www.datacenterdynamics.com/en/news/google-to-build-new-uk-data-center-campus-in-hertfordshire/

https://opensource.com/resources/what-open-source

https://en.wikipedia.org/wiki/Open-source_software

https://www.cloudways.com/blog/horizontal-vs-vertical-scaling/

https://learn.microsoft.com/en-us/devops/develop/what-is-continuous-integration

https://learn.microsoft.com/en-us/devops/deliver/what-is-continuous-delivery

https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines

https://azureossd.github.io/2023/04/17/Troubleshooting-Container-didnt-respond-to-HTTP-pings-failing-to-start-site/