Evaluating the Effect of multi-tenancy Patterns in Containerized Cloud-hosted Content Management System

***Abstract*—Multi-tenancy in cloud computing describes the extent to which resources can be shared while guaranteeing isolation among components (tenants) using these resources. There are three multi-tenancy patterns: shared, tenant-isolated and dedicated component patterns. These patterns have not previously been formally specified. So how do we choose an appropriate multi-tenancy pattern for a multi-tenant application? To address this question, we have created a formalized description of each multi-tenancy pattern in Z language. We formalize the multi-tenancy pattern using fixed semantics, firstly, to verify each of the patterns, secondly, to provide a precise interpretation of the pattern and finally, to choose a suitable multi-tenancy pattern for a multi-tenant application. We then empirically evaluate each pattern using the data-tier of a cloud hosted distributed content management application, WordPress, deployed in a Docker container. Experimental results show that the dedicated pattern performed best with varying tenant needs while shared and tenant-isolated patterns performed variably the same depending on how much data were involved. Based on the empirical evaluation, we provide a selection algorithm to choose suitable multi-tenancy pattern for software deployment.**

I. Introduction

Cloud computing uses the Internet to distribute computing resources as a utility [1] [2] [3]. It provides scalable: resource provisioning, IT infrastructure, development platforms, data storage and software applications [4] [5] [6]. In recent years, there has been an increase in the number of users served by software applications on cloud [7] commonly known as Software-as-a-service (SaaS).

SaaS, a cloud computing model for accessing software applications over the Internet [4], it eliminates installation on client infrastructures, centralises maintenance, reduces total cost of ownership [8] and are easy to use since they already come with best practices. However, to effectively support SaaS users, a multi-tenant application is required, where groups of users are classified as tenant [7] and multiple tenants shares application and database instances of a SaaS. A multi-tenant application is actualised through a technology called multi-tenancy.

Multi-tenancy is a software architecture where a single instance of software runs on a server and is used to serve multiple groups of users, called tenants [7]. Tenants share common access to the software instance and are granted specific access privileges [7]. Multi-tenancy improves resource utilisation and the cost of service provision can be spread across multiple tenants using a shared application and database instances [7]. It also centralises application and database instances thereby simplifying software upgrades and deployment [6].

Three multi-tenancy deployment patterns have been defined [9] [10]: shared, tenant-isolated and dedicated pattern. In this research we formally specify each of these patterns for the first time. This leads us to the research question: which multi-tenancy pattern is suitable for a multi-tenant application? We then experimentally evaluate each multi-tenancy pattern by using a case study cloud hosted distributed content management application, WordPress, deployed in a Docker container. Furthermore, we present a novel selection algorithm to choose suitable multi-tenancy pattern for software deployment.

The rest of this paper is structured as follows. Section 2 describes the three variants of multi-tenancy patterns. Section 3 reviews related work on multi-tenancy in cloud environment. Section 4 describes our experimental setup, results and comparison of performance with WordPress application. Section 5 provides the algorithm for choosing the right pattern. Section 6 concludes the paper, provides recommendations on the right multi-tenancy pattern to use based on users' requirements, and also talks about the scope for further work.

II. Variants of Multi-tenancy Patterns

This section describes the three multi-tenancy patterns.

A. Shared Pattern

shared pattern is the first variant of the multi-tenancy pattern. It is the basic minimum requirement for resource sharing in a SaaS application. This pattern is implemented to share resources by serving different tenants without maintaining a notion of tenant itself, and because of this reason, a tenant activity can influence another tenants' functionality especially data intrusion [11] [10].

This pattern makes tenants to share the same database instance, schema and tables. Each tenant is identified using a variable called the tenant ID, this allows the database to group users with the same tenant ID as belonging to the same tenant.

We present shared pattern using Z language in Figures 1 - 5. The formal definition describes shared pattern in terms of the basic CRUD operations that can be performed on it. Figure 1 describes how shared pattern should be created in the data-tier level of a SaaS application. Figure 2 shows how a new table item called tenant item would be added to the shared database; it checks if a database and the table exists before

$$
\begin{array}{|l}
\textit{SharedDB} \\
\hline
known : \mathbb{P}\, SysDBName \\
SharedDBItem : SysDBName \pfun SharedTableItem \\
\pfun TenantID \\
\hline
known = \mathrm{dom}\, SharedDBItem \\
\hline
\end{array}
$$

Figure 1. Z Representation of the Overall Shared Pattern

$$
\begin{array}{|l}
\textit{SharedDB.AddTenantItem} \\
\hline
\Delta SharedDB \\
name? : SysDBName \\
\hline
name? \notin known \\
SharedDB' = name? \cup name? \cup \\
SharedTableName? \cup TenantID? \\
\hline
\end{array}
$$

Figure 2. Z Representation of Creating Tenant in Shared Pattern

adding in any data. Figure 3 shows how data is read from tables in the database, it verifies database, table and tenant details before data is read. Figure 4 shows how data can be updated in this pattern, it verifies database, table and tenant details; it also check that the data to be updated already exist. Figure 5 shows how tenant data can be deleted.

B. Tenant-Isolated Pattern

Tenant-isolated pattern is the second variant of the multi-tenancy pattern. It helps to address a large number of customers, effectively utilize resources among these customers, and in turn leverage economies of scale.

It enables sharing of resources with intermediate levels of performance, security, privacy and resource overheads. This means influences of tenants regarding assured performance, available storage capacity and accessibility can be avoided. It can be used for authentication isolation, access control isolation, information protection isolation, performance, fault and administration isolation.

$$
\begin{array}{|l}
\textit{SharedDB.ReadTenantItem} \\
\hline
TableName? : SysDBTableName \\
SharedTableItem? : SharedDB \cup \\
TenantID? : SharedDB \\
\hline
SharedDB' = SharedDb : SharedDb' \bullet \\
if\ SharedDB.SysDBName = SharedDB?.SysDBName \\
then\ SharedDB? \\
else\ SharedDB \\
\hline
\end{array}
$$

Figure 3. Z Representation of Reading Tenant's Data in Shared Pattern

$$
\begin{array}{|l}
\textit{SharedDB.UpdateTenantItem} \\
\hline
\Delta SharedDB \\
name? : SysDBName \\
SysDBName? : SharedDB \cup \\
SharedTableItem? : SharedDB \cup \\
TenantID? : SharedDB \\
\hline
SharedDB' = SharedDb : SharedDb' \bullet \\
if\ SharedDB.SysDBName = SharedDB?.SysDBName \\
then\ SharedDB? \\
else\ SharedDB \\
\hline
\end{array}
$$

Figure 4. Z Representation for Data Update in Shared Pattern

$$
\begin{array}{|l}
\textit{SharedDB.DeleteTenantItem} \\
\hline
\Delta SharedDB \\
name? : SysDBName \\
\hline
name? \in known \\
SharedDB' = SharedDB : SharedDB' \mid \\
name? \ominus \{name? \mapsto name?\} \\
\hline
\end{array}
$$

Figure 5. Z Representation of Deleting Data in Shared Pattern

Tenant-isolated implementation in the data-tier of a SaaS features sharing of the same database instance, while database schema and tables are dedicated to each tenant.

We present tenant-isolated pattern using Z language in Figures 6 - 7. Figure 6 depicts how a tenant-isolated database can be created while Figure 7 shows how tenant and tenant items can be added to a tenant-isolated database.

$$
\begin{array}{|l}
\textit{TenantIsolatedDB} \\
\hline
known : \mathbb{P}\, SysDBName \\
TenantDBItem : SysDBName \pfun TenantIsolatedTableItem \\
\pfun SchemaID \\
\hline
known = \mathrm{dom}\, TenantIsolatedDBItem \\
\hline
\end{array}
$$

Figure 6. Z Representation of Tenant-Isolated Pattern

$$
\begin{array}{|l}
\textit{TenantIsolated.AddTenantItem} \\
\hline
\Delta TenantIsolatedDB \\
name? : SysDBName \\
\hline
TenantName? \notin known \\
TenantIsolatedDB' = name? \cup name? \cup \\
SchemaID \pfun TenantIsolatedTableName? \cup SchemaID? \\
\hline
\end{array}
$$

Figure 7. Z Representation of Creating New Tenant in Tenant-Isolated Pattern

DedicatedDB

$known : \mathbb{P}\, SysDBName$

$DedicatedDBItem : SysDBName \pfun DedicatedTableItem \pfun SysDBName$

$known = \mathrm{dom}\, DedicatedDBItem$

Figure 8. Z Representation of Dedicated Pattern

DedicatedDB.AddTenantItem

$\Delta DedicatedDB$

$name? : DedicatedDBName$

$TableName? \notin known$

$TenantIsolatedDB' = name? \cup name? \cup SchemaID \pfun TenantIsolatedTableName? \cup SchemaID?$

Figure 9. Z Representation of Creating Tenants and Data in Dedicated Pattern

C. Dedicated Pattern

The third variant of multi-tenancy pattern is the Dedicated pattern. This pattern provides exclusive access to components that features critical functionality [10] while other components can still be shared. The motivation behind this pattern is the need to fulfill some data and application protection rule that requires that data or application be kept secured without being compromised by other tenants' data or application.

On the data-tier of a SaaS, a dedicated database instance is allocated to the tenant to support critical component of the application, other parts of the application data can be safely stored in a multi-tenant implementation of a database.

The Z representation of this pattern is depicted in Figures 8 - 10. Figure 8 depicts how an overall database using the dedicated pattern should be created. It defines an application specific database name with dedicated table items that contains zero or more rows. Figure 9 describes how tenant and tenants' data are added to a dedicated database. It checks that the database and table exists, and checks for changes in the database when tenants' data are added. Figure 10 represents an update of tenants' data in the database. It checks for changes in the tenants' table when an update is made.

III. Related Work

The multi-tenancy concept has birthed several researches on enabling an effective use of this architectural pattern.

DedicatedDB.UpdateTenantItem

$\Delta DedicatedDB$

$name? : DedicatedDBName$

Figure 10. Z Representation of Deleting Data in Dedicated Pattern

Previous work on multi-tenancy architecture have focused on performance, data isolation, and robustness of multi-tenant systems. However these research lacked a formal description of the multi-tenancy pattern itself, which is suppose to be the foundation on which the architecture can be effectively used and built. This paper aim to close this gap by formally describing the multi-tenancy pattern using Z language and then present a selection algorithm to choose suitable multi-tenancy pattern when building a multi-tenant application.

Multi-tenancy, an approach to share an application instance between multiple tenants is a key architectural pattern for a multi-tenant SaaS application. In general, it yields cost benefits in cloud environments and it is most efficient on level of application instances [6]. [6] explored the challenges of performance isolation in the context of multi-tenant SaaS applications and they proposed a middleware architecture to enforce performance isolation based on the tenant-specific SLAs using a tenant aware profiler and a scheduler. They identified the greatest challenge of performance isolation as being able to offer application-level multi-tenancy while performance isolation between different tenants is not hampered.The experiment in this research reviewed mainly performance isolation and did not compare the performance of the different multi-tenancy patterns.

Further research was done by [12] by describing multi-tenancy as a key enabler in building cloud middleware that maximizes sharing and support of application. Their primary contributions are motivating work-flow multi-tenancy, design and architecture of a multi-tenant work-flow engine that enables multiple users to run their work-flow securely within the same work-flow engine instance and a performance evaluation of the architecture. The evaluation carried out in this contest did not compare the variants of multi-tenant architecture, it only evaluated the multi-tenancy pattern used in the work-flow application.

[13] evaluated the robustness of cloud-based system after being inspired by Ecology. Their solution is built upon an analogy between species extinction and component failures. They identified three robustness indicators namely: overall robustness, the most sensitive components and the most threatening failure sequence. They also built an app to show how their identified robustness indicators can help to sort out architectural decisions regarding robustness of which multi-tenancy is key.

[14] modeled customizable SaaS applications using feature modeling. They identified one disadvantage of multi-tenancy as being difficult to create customizable applications. Based on this premise,they describe an approach for the development and management of highly customizable multi-tenant cloud applications. This approach applied software product line engineering techniques and dynamic feature placement algorithm to applications composed of multiple interacting components. They evaluated the results of this algorithm and concluded that it helps to manage customized applications up to 77% of the time.

[15] developed a highly adaptable and scalable monitoring

architecture for multi-tenant Clouds. They identified the need for cloud administrators to design better cloud provisioning strategies so that they can avoid SLA violations while they provide services to multiple tenants. Their solution is a distributed architecture for resource management and monitoring in clouds, a distributed cloud monitoring architecture to disseminate resource monitoring information while keeping a low overhead. They also reported experimental results to assess the architecture and quantitatively compare it with a selection of other cloud monitoring tools similar to their implementation, they were able to show that their implementation outperforms these tools and introduces very limited overhead when monitoring the applications.

[8] Developed a multi-tenancy performance benchmark for web application platforms. It highlights lack of performance guarantee as a major obstacle to the adoption of cloud computing and Mutli-tenant application in particular. In an attempt to solve this issue, they present an extended version of an existing accepted application benchmark(TPC benchmark) by adding support for the Multi-tenant platform features. The extended benchmark focuses on evaluating maximum throughput and the amount of tenants that can be served by a platform. This in turn will help cloud providers to provide realistic performance guarantee and can also help them monitor and scale performance of any multi-tenant application they on their platform.

[16] Identified performance isolation as key requirement for application level multi-tenant sharing hosting environment. This will in-turn allow centralization of infrastructure in locations with lower cost and on the long run reduce operational and delivery costs. They proposed a novel approach to dynamically estimate application level CPU consumption based on Kalman filter. They implemented their research using three approaches for multi-tenant deployment: shared infrastructure, shared middle-ware and shared application. Having implemented their tool on the three different sharing levels, they concluded that controlling resource consumption such as CPU and memory of each tenant is extremely valuable for performance isolation in all the three levels of multi-tenant deployment approaches they have earlier identified.

[17] Explored different kinds of typical multi-tenant data tier implementation patterns on aspects of isolation, security, customization and scalability. They also evaluated the performance of these patterns through a series of experiments and summarize a set of valuable conclusion and best practices on how to design a multi-tenant data model.

[18] evaluated the degree of isolation among tenants using a component based approach to multi-tenancy through re-routing. This evaluation followed an empirical studies approach, where they implemented the three identified multi-tenancy patterns in a multi-tenant component of Hudson's file system. Their result provides extensive report on each pattern and the degree of Isolation it is able to give. They were also able to give recommendation to software architects on choosing the right pattern that implements a required isolation in cloud hosted version control system.

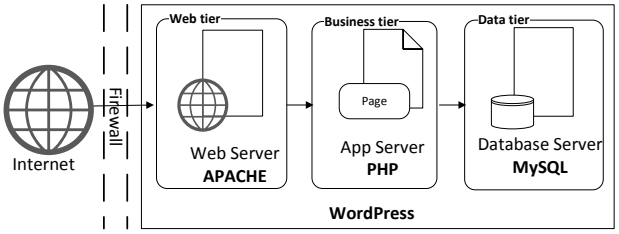


Figure 11. Architectural Setup of WordPress

Having considered these research work, our research differs from them because this paper's focus is to create a foundation on which multi-tenancy pattern can be used.

IV. Methodology

A. Case Study

We implemented the multi-tenancy pattern in the data-tier of a cloud hosted dockerized WordPress application. We choose WordPress as the benchmark application because it is an open source and distributed content management system that is widely used and accepted. It powers 27 percent of the Internet through websites, and blogs and is usually offered freely on two main levels: content management software with default settings, features and customisable core; or as an SaaS via wordpress.com [19].

1) WordPress Installation: WordPress is developed in PhP and backed up by MySQL database. It is usually deployed on a web server such as Apache and accessed via a web browser as seen in figure 11.

This research used the first distributed level of WordPress to evaluate the three multi-tenancy patterns, because it opens up more control and flexibility on modifying different parameters to suit our experimental setup. To install and dockerize WordPress, docker and it's accompany tools were installed on three virtual machines, docker-compose was used to pull and modify docker images of WordPress and MySQL database as seen in Figure 12, and docker command was then used to run containers from these images.

2) Multi-tenancy Pattern Implementation: We examined WordPress' business logic to identify its default data-tier pattern and found the dedicated pattern to be its default data-tier implementation. So, we created three separate docker images of WordPress, each image represents a WordPress instance with one of the multi-tenancy pattern implemented in it. We implemented the tenant-isolated and shared pattern by changing the config.php and functions.php file of two WordPress instances. This creates database tables that corresponds to the formal description of the multi-tenancy patterns in Figures 1 - 10 and also adjust the business logic to save data in the format represented by the database. The diagram in Figure 13 shows how some database tables are setup to represent a tenant-isolated implementation in WordPress.

3) Experimental Setup: The goal of this experiment is to evaluate the performance of the three multi-tenancy patterns in WordPress. Three virtual machines with these configurations:

```
wordpress:
          image: guss77/docker-wordpress-multisite
          links:
           - wordpress_db:mysql
          ports:
           - 8080:80
 volumes:
           - ~/wordpress/wp_html_wpdata  :/var/www/html
wordpress_db:
           image: mariadb
 volumes:
          - ~/data/:/var/lib/mysql
 environment:
          MYSQL_ROOT_PASSWORD: *****
          MYSQL_DATABASE: wordpress
          MYSQL_PASSWORD: *****
```

Figure 12. Docker-Compose for Pulling and Running WordPress - Tenant-Isolated Pattern Implementation

500GB HDD, 3GB memory, and Ubuntu 16.04 LTS operating system were setup and the following tools were installed on each of them: WordPress 4.8.2, MySQL 5.7.21, Apache Jmeter 3.2, docker 1.8.0 and Apache web server.

B. MySQL

MySQL is an open source relational database management system (RDBMS) that use the structured query language to add, access and manage data in a database [20]. It is the default RDBMS used by WordPress.

C. Docker

Docker is a light weight virtualization concept that package software into a standardized units for development, shipment and deployment called container [21]. Containerized software applications will always run the same regardless of the environment and infrastructure. We use docker to package WordPress so as to leverage virtualisation technique of cloud.

D. Apache Web Server

Apache web server is a free and open source cross platform HTTP server for hosting and/or serving web sites [22]. We used Apache web server to host the WordPress installation.

E. Apache Jmeter

Apache Jmeter is an open source software developed in Java to load test functional behaviour and measure performance of static and dynamic resources [23]. We used Apache Jmeter to write scripts that load test WordPress, these scripts contains Jmeter samplers and parameters that translates to web pages, actions performed, number of users performing the actions and how they are performed in WordPress. Three scripts each representing a load test script for each of the multi-tenancy patterns were created. The script simulates group of users (tenants) sending requests to create blog-posts in WordPress. Requests sent is termed a business process as shown in table I.

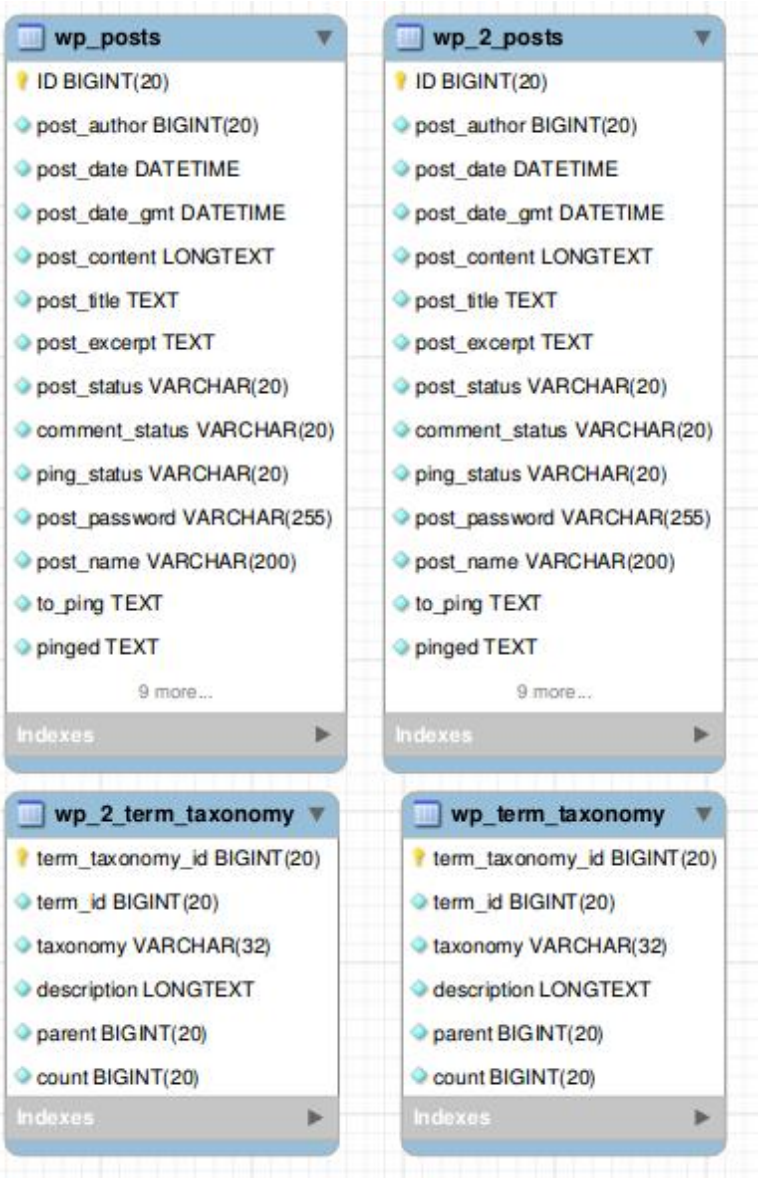


Figure 13. Tenant-Isolated Database Implementation of WordPress

Table I
BUSINESS PROCESS 2

S/N	Business Process	BytesofData
1	Login	1110.33
2	Create Post	2220.67
4	Log out	1110.33

Tenants were grouped into sets of 20 users and the scripts increased the number of users progressively till it reached 200. Each request from a user sends data of not less than 5kb at once, this data contains the content of a blog post such as texts, tables, pictures and links.

We also setup three database instances as docker containers, each instance represents one of the multi-tenancy patterns and backs up the WordPress instance for that variant.

The overall experimental setup as shown in Figure 14

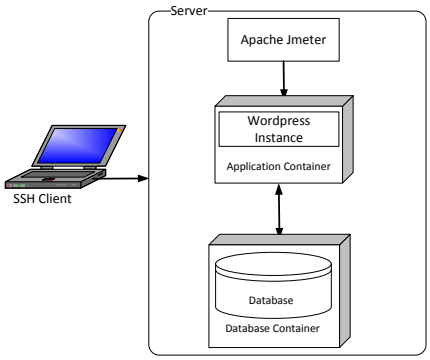


Figure 14. Architecture of Experimental Setup

represents how each WordPress instance is setup to implement the multi-tenancy pattern.

V. Result

We ran the experiment to measure response times of the business process carried out on each instances of our WordPress installation. We averaged 5 runs of each group of user requests to determine the groups' average response time of the business process for each experimental setup of WordPress. Equation 1 depicts how the average response time is calculated for each tenant:

$$Avg.ResponseTimeforNruns = \frac{\sum_{N}^{1}(\frac{\sum(E)}{T_n})}{N} \quad (1)$$

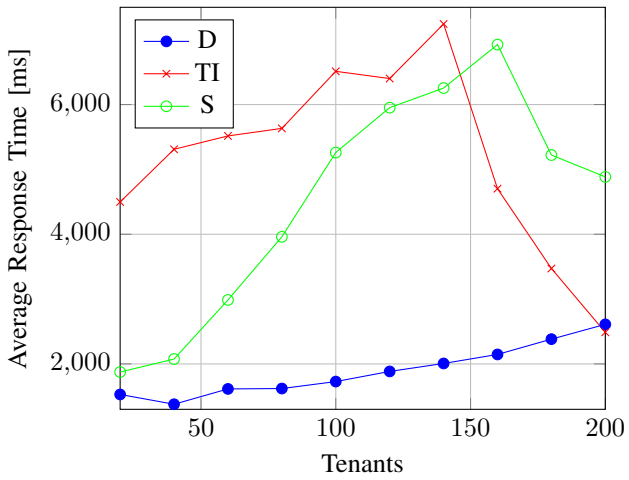
N = Number of runs, T_n = Number of tenants, E = Elapsed time

Figure 15 depicts the average response time of creating blog-posts in the three experimental setup of WordPress implementing the multi-tenancy pattern. These response times were as a result of increase in the number of tenants that were being served; the number of tenants increased from 20 to 200 in each of the three experimental setup.

The average response time for the dedicated pattern ranged from 1 minute 20 seconds to 2 minutes 30 seconds to create blog-posts. Average response time for tenant-isolated pattern ranged from 2 minutes 10 seconds to 7 minutes 20 seconds. Average response time for shared component pattern ranged from 1 minute 35 seconds to 6 minutes 45 seconds. The result in Figure 15 also showed that the average response times of shared and tenant-isolated pattern reached a peak at 6 minutes 45 seconds and 7 minutes 20 seconds respectively before it dropped. The drop in the response time reflects the presence of errors in the number of requests being handled by WordPress.

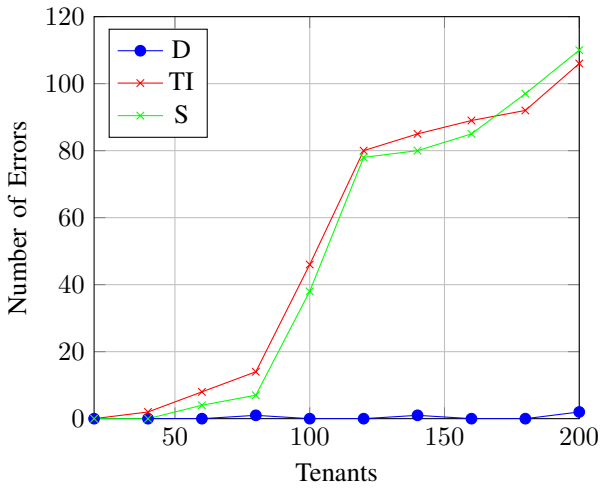
Figure 16 shows the number of failed requests that occurred while evaluating the multi-tenancy patterns in each of the experimental setup of WordPress. We see an increase in number of failed requests when the average response time peaked for both tenant-isolated and shared pattern.

Figure 16 shows the number of errors encountered during the experiments for each of the pattern.



D - Dedicated Pattern
TI - Tenant-Isolated Pattern
S - Shared Pattern

Figure 15. Average Response Time Graph for the three Multi-tenancy Pattern when Creating Blog-Posts



D - Dedicated Pattern Errors
TI - Tenant-Isolated Pattern Errors
S - Shared Pattern Errors

Figure 16. Errors from Request in Multi-tenancy Pattern Implementation when Creating Blog-Posts

VI. Discussion

In other to address our research question: which multi-tenancy pattern is suitable for a multi-tenant application? we analyse our experimental results. The experimental results show that dedicated pattern performed better in WordPress than Shared and tenant-isolated patterns under varying number of tenants and can handle 140 tenants creating blogs on WordPress within 2 minutes.

Shared and tenant-isolated patterns performed almost the same and can handle 140 tenants in 6 minutes 30 seconds before their performances starts to dwindle. The shared and

tenant-isolated patterns used longer times to handle tenants' requests, because tenants of this pattern implementation refer to multiple groups of users that do not belong to the same wider group.

So what does this mean for WordPress? From one perspective, dedicated pattern suits WordPress applications used within an organisation where users are all grouped as a tenant, however from another perspective, tenant-isolated pattern suits WordPress applications in two different instances: tenants are groups of users within an organisation or tenants are different groups of users that are not part of the same organisation. The tenant-isolated pattern approach of sharing resources will allow more groups of tenants to share the same WordPress instance with different customisation and database schema. The shared pattern performs well in WordPress when tenants do not exceed 100 users; however we recommend that this setup be used for handling less critical data such as setting up user data of a SaaS application.

A. Strength and weakness of the Selected Multi-tenancy Pattern

The three multi-tenancy patterns represent the full range of multi-tenant needs in a SaaS. The shared pattern reduces the use of resources to the barest minimum, however this is at the expense of data, performance and process isolation [11], hence it should be used when data and performance capability is not a major concern.

Tenant-isolated pattern represents a compromised implementation between the shared and dedicated patterns [10]. This pattern promotes privacy and data security as tenants will not have their data mixed with other tenants' data. However, there is an increase in the cost of database connection and reduced utilisation of resources as compared to the shared component pattern. This pattern is usually featured in most multi-tenant application database. Example of this is a payroll application used by multiple divisions in a large company, where each division is implemented as a tenant-isolated component. Another example is its usage in the multi-site database implementation of WordPress, where each site has its own schema and tables.

Dedicated pattern features the highest form of privacy and data security; however, this is at the expense of high usage of available resources and nothing to less benefit from economies of scale. This pattern should be used when there is sufficient reason for data not to be mixed with other tenants' data.

B. Performance and Errors

The performance of the three multi-tenancy patterns differed under varying tenants' needs, with their average response times steadily increasing as the number of tenants increased. Dedicated pattern performed best with a maximum average response time of 2 minutes 30 seconds. However, Figure 16 shows some slight errors while using this pattern, though the amount of errors encountered are quite negligible and can be attributed to the fact that connections speed may vary at specific times.

Tenant-isolated pattern implementation performed well considering the tenants' complexity it handled as explained in section VI. Figure 15 shows that the average response time peaked when the number of users was 140 and then dropped afterward. The drop in average response time indicates that WordPress was not able to handle all the requests from Jmeter, hence it dealt with a reduced number of requests and this reflects in the number of errors generated (requests not handled) as shown in Figure 16.

The shared pattern performed variably the same as tenant-isolated pattern, however the time at which its average response time peaked and the amount of errors recorded as compared with tenant-isolated pattern shows that tenant-isolated pattern performed better than shared pattern by the time the number of tenants exceeded 120 because Figure 15 showed that more requests were being handled by tenant-isolated pattern at that point.

C. Recommendation

The experiment helped to identify multi-tenancy pattern that suits different cloud hosted content management application setup. Figure 1 is a selection algorithm that will help to choose a suitable multi-tenancy pattern in a multi-tenant content management application

Algorithm 1: Algorithm to Choose Multi-tenancy Pattern

```
if SaaS then
    if data sent per request ≤ 5kb then
        if data privacy ≠ important then
            Shared Pattern ;
else if data sent per request > 5kb and
 data privacy ≠ important then
    Tenant-isolated pattern ;
else
    Dedicated Pattern ;
```

VII. Conclusion

Multi-tenancy is an important and frequently used architecture in SaaS. It improves resource utilisation, reduces the effort in deploying applications on cloud and total cost of ownership. However, choosing the correct multi-tenancy pattern for a multi-tenant application is a challenge. To solve this challenge, We present the multi-tenancy patterns using Z, we evaluated the performance of the three multi-tenancy patterns in order to access its suitability in WordPress and we provide a novel selection algorithm to choose a suitable multi-tenancy pattern for a multi-tenant content management system.

Our experimental setup used docker to package WordPress and its dependencies into containers that were deployed onto an Open stack cloud, and Jmeter to create experimental harness to test the performance of the three multi-tenancy patterns in WordPress. We empirically evaluate the performance of the three multi-tenancy patterns by measuring the throughput

of a commonly used business process - blog creation, in WordPress.

We found that the dedicated pattern performed best even with varying user needs, while tenant-isolated and shared patterns performed variably same depending on how much data were involved.

In the future, we aim to extend this research to evaluate the effect of multi-tenancy patterns in other containerized cloud deployment scenarios such as cloud storage systems.

REFERENCES

[1] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *2008 Grid Computing Environments Workshop*, Nov 2008, pp. 1–10.

[2] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Dec. 2008. [Online]. Available: http://doi.acm.org.salford.idm.oclc.org/10.1145/1496091.1496100

[3] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," *INC, IMS and IDC*, pp. 44–51, 2009.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep., Feb 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[5] B. P. Rimal and M. Maier, "Workflow scheduling in multi-tenant cloud computing environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 290–304, Jan 2017.

[6] S. Walraven, T. Monheim, E. Truyen, and W. Joosen, "Towards performance isolation in multi-tenant saas applications," in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, ser. MW4NG '12. New York, NY, USA: ACM, 2012, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/2405178.2405184

[7] L. Fan, B. Gao, Z. Wang, W. An, and Y. Wang, "A semi-automatic approach of transforming applications to be multi-tenancy enabled," *IEEE Transactions on Services Computing*, vol. 9, no. 2, pp. 227–240, March 2016.

[8] R. Krebs, A. Wert, and S. Kounev, "Multi-tenancy performance benchmark for web application platforms," in *Proceedings of the 13th International Conference on Web Engineering*, ser. ICWE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 424–438. [Online]. Available: http://dx.doi.org/10.1002/spe.2320

[9] C. F. F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, "Cloud computing patterns," *Springer, Wien. doi*, vol. 10, pp. 978–3, 2014.

[10] AA, "Omitted for blind review," in *Omitted for blind review*, Nov 2015, pp. 53–58.

[11] ——, "Omitted for blind review," in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 1, 2015.

[12] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana, "A multi-tenant architecture for business process executions," in *2011 IEEE International Conference on Web Services*, July 2011, pp. 121–128.

[13] F. Chauvel, H. Song, N. Ferry, and F. Fleurey, "Evaluating robustness of cloud-based systems," *Journal of Cloud Computing*, vol. 4, no. 1, p. 18, 2015. [Online]. Available: http://dx.doi.org/10.1186/s13677-015-0043-7

[14] H. Moens, B. Dhoedt, and F. D. Turck, "Allocating resources for customizable multi-tenant applications in clouds using dynamic feature placement," *Future Generation Computer Systems*, vol. 53, pp. 63 – 76, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X15002010

[15] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, and L. Foschini, "Dargos: A highly adaptable and scalable monitoring architecture for multi-tenant clouds," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2041 – 2056, 2013, including Special sections: Advanced Cloud Monitoring Systems and The fourth IEEE International Conference on e-Science Applications and Tools and Cluster, Grid, and Cloud Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X13000824

[16] W. Wang, X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong, "Application-level cpu consumption estimation: Towards performance isolation of multi-tenancy web applications," in *2012 IEEE Fifth International Conference on Cloud Computing*, June 2012, pp. 439–446.

[17] Z. H. Wang, C. J. Guo, B. Gao, W. Sun, Z. Zhang, and W. H. An, "A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing," in *2008 IEEE International Conference on e-Business Engineering*, Oct 2008, pp. 94–101.

[18] L. C. Ochei, A. Petrovski, and J. M. Bass, "Evaluating degrees of tenant isolation in multitenancy patterns: A case study of cloud-hosted version control system (vcs)," in *2015 International Conference on Information Society (i-Society)*, Nov 2015, pp. 59–66.

[19] WordPress. (2002) Wordpress features. [Online]. Available: https://codex.wordpress.org/WordPress

[20] Siteground. (2004-2017) What is mysql. [Online]. Available: https://www.siteground.co.uk/tutorials/php-mysql/mysql.htm

[21] Docker. (2017) What is docker. [Online]. Available: https://www.docker.com/what-docker

[22] Wikipedia. (2017) Apache http server. [Online]. Available: https://en.wikipedia.org/wiki/Apache_HTTP_Server

[23] A. S. Foundation. (1999-2017) Apache jmeter. [Online]. Available: http://jmeter.apache.org/