



UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

Doble grado en Ingeniería Informática y Matemáticas

MW Store: facilitador genérico para proveedores de servicios con multi-inquilinato en una infraestructura Cloud

Autor

Rubén Morales Pérez

Director

Manuel Isidoro Capel Tuñón

Departamento de Lenguajes y Sistemas Informáticos

Contents

1	Introduction	4
1.1	Classification	5
2	Security	7
2.1	Trusted Database Connections	8
2.2	Secure Database Tables	8
2.3	Tenant Data Encryption	10
3	Extensibility	10
3.1	Multitenancy customization	11
3.2	SaaS architecture approaches	13
3.3	Deployments	13
3.4	Software regressions	13
3.4.1	Automatic testing	13
3.4.2	Tests detection	15
3.5	Observability and monitoring	15
4	Performance	15
4.1	Database	15
4.2	Data scalability patterns	15
4.2.1	Concurrent data accesses	16
4.3	Cache	16
4.4	Quality of Service in Cloud Computing	16
5	Petri nets	16
5.1	Concurrent systems validation	16
6	Use case: e-commerce platform	16
6.1	Coverability Trees using Petri Nets	16
6.1.1	Availability	16
6.1.2	Vivacity?	16
6.1.3	Security	16

1 Introduction

The influence of cloud computing and Software-as-a-Service (SaaS) is growing in enterprise software. SaaS is frequently offered in a multi-tenant paradigm, where multiple customers (tenants) share resources such as software and hardware, without necessarily sharing data [3]. Each tenant will be able to store the data for their own customers. Multi-tenancy allows customization of the software according to the requirements of different customers. Multi-tenancy is applicable to different levels in a system's architecture and there are several architectural approaches to define it. It's possible to refer to multi-tenancy using the whole picture of a system or increasing the granularity (i.e. database level). Several core principles in multi-tenancy will be covered: data isolation, security, concurrent data access, scalability and performance.

Definition 1. NIST definition of cloud computing Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [9]

In general, cloud computing reduces business risks (such as hardware failures) and maintenance expenses by outsourcing the service infrastructure to the infrastructure providers [9]. Virtualization allows cloud computing to pool computing resources from clusters of servers and distribute them dynamically to the applications. Dynamic resource provisioning can be reactive to immediate demand fluctuations or proactive, allocating resources before they are needed using predicted demand [9]. The architecture of cloud computing (Figure 1) can be described in four layers [9]:

- Hardware layer: handle the physical resources of the cloud, servers, routers, switches, power and cooling systems.
- Infrastructure/virtualization layer: pool of storage and computing resources. Infrastructure as a Service (IaaS) refers to the on-demand provisioning of infrastructural resources like virtual machines.
- Platform layer: operating systems and application frameworks. It's main purpose is to minimize the effort to deploy applications on virtual machine containers. Platform as a Service (PaaS) refers to providing these resources on demand, examples of PaaS providers could be Google App Engine and Microsoft Windows Azure.
- Application layer: the actual cloud applications. Software as a Service (SaaS) provides on-demand applications over the Internet using web interfaces.

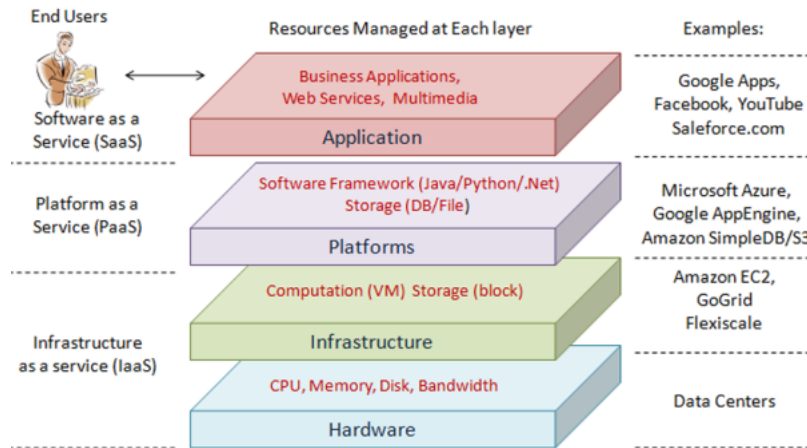


Figure 1: Cloud computing architecture [9]

From an economic perspective, clouds are implemented in large data centers, achieving economy-of-scale and high manageability [9]. The main drawbacks of this approach is the high energy expense and high initial investment. On the other hand, small data centers does not consume so much power and are cheaper to build and can be distributed geographically easier than large data centers.

1.1 Classification

From the isolated approaches to more shared data [1], multi-tenant data classification is an continuum:

- **Separate databases.** It is the simplest approach to data isolation. Generally, computing resources and application code are shared between all the tenants, but data is logically separated. It adds an extra database data security layer, but it tends to higher costs, limiting the number of tenants to the number of databases that the server can support.
- **Shared Database, Separate Schemas.** Each tenant has its own set of tables grouped into a separated schema. This approach still allows for database permissions layer and it has lower costs that separate databases. It could be a good approach for applications with a small number of database tables.
- **Shared Database, Shared Schema.** Each table can include records from multiple tenants, a table column associates each record with the appropriate tenant. Data isolation is achieved through virtual isolation. This approach has the lowest hardware and backup costs, allowing the largest number of tenants per database server.
- **Mixed approaches.** It is possible to have separate databases/schemas for specific customer(s) and share schemas for other customers. This approach could fit the requirements of certain customers but the extra complexity must be justified.

Some customers will have strongest data isolation requirements than others. Separate databases in practice would simply add an extra security layer due to more granularity regarding database access permissions. Some key points to decide which configuration is more appropriate for your business case are:



- Number of tenants. The order of magnitude in number of tenants might prevent scalability with separate databases.
- Database size per tenant. Huge amounts of data per tenant would make shared approaches more inconvenient. Strategies like grouping tenants by geographical distribution could be considered to minimize this effect.
- Configurability, possibility of tailored tenant services. Separate databases would allow easier configurability per tenant, creating custom tables and columns in the database. The business model is relevant here because more shared approaches would make feature toggle straight forward, allowing to turn on/off certain features on the fly.

On the other hand, just because a system is advertised as SaaS in the cloud does not mean it holds its benefits (savings, security, flexibility, disaster recovery, scalability, access to automatic updates, ...). Microsoft describes an incremental SaaS maturity along three dimensions: configurability, multi-tenancy and scalability [4]. The axis of the maturity model combines the service component axis (core features of structuring software business) and the maturity levels (Figure 2):

- Ad Hoc Level: dedicated database and schema without content sharing and multitenancy. It is handled with different applications on web interfaces for each user. With respect to the Service-level agreement (SLA), it depends on separated contracts reflecting the requirements for each user.
- Standardization Level: attempts to provide shared service. It shares the database using dedicated schema with configurable single tenants, allowing users to build their service model.
- Integration Level: multi tenant environment with shared database and schema. There is a predefined shared instance and configurable options for each tenant. The business layer focuses on measurable SLA adaptations.
- Virtualization Level: focus on scalability and availability. The data layer uses distributed computing. The virtualized system uses load-balancing across several service modules. User requirements are handled in the business process (instead of code customizations) and optimized SLA policies by measuring the amount of service used per customer

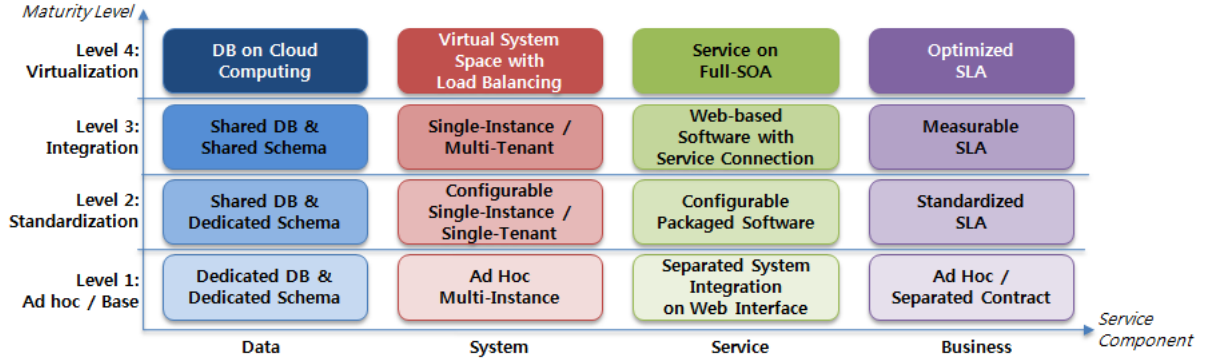


Figure 2: SaaS Maturity Model [4]

The layered architecture of cloud computing allows a PaaS provider to run on top of an IaaS provider. In practice, IaaS and PaaS belong to the same organization and are often called the infrastructure providers or cloud providers[9].

Some service providers optimize operation costs, while others prefer high reliability and security. There are several types of clouds available:

- Public clouds: offering resources to the general public.
- Private clouds (internal clouds): exclusive for a given organization. They offer more control over performance, reliability and security at a higher cost than public clouds.
- Hybrid clouds: a combination of public and private clouds, the separation between public and private cloud components must be designed in advance.
- Virtual private clouds (VPC): a platform running on top of public clouds. It virtualizes servers, applications and the underlying communication network through virtual private networks.

2 Security

SaaS forces potential customers to partially shift control of their business data to a third party. Security is one the basic concerns that needs to be taken into account in order to gain customer trust. In risk analysis, multiple defense levels against both internal and external potential threats will increase the overall security. The system is only as strong as the weakest link in the chain.

SaaS applications provide an extra layer of security allowing to handle security in the back end, although extra layers of defense in the front end are a nice to have. In desktop applications the code could potentially be reverse-engineering and modified to meet certain goals. Security in the front end is a nice to have but we must take into account that any security layer implemented in the front end could potentially be bypassed.

Security must be considered by design, not to rely on security through obscurity. There are several security patterns (agnostic to the database management system) available for each data isolation

approach [1]. In the particular scenario of multi-tenancy, these security approaches should be taken into account to provide security at different levels:

- Filtering: use an intermediary layer between a tenant and a data source, allowing access only to their tenant data (data isolation). In the code, this can be handled restricting the persistence layer access to certain database access objects filtering automatically the tenant data. Another safer possibility would be to use query interceptors to restrict database access to other tenants' data. The tenant id could be required for each request to the back end. For additional security, this tenant id can be stored in a temporal token previously generated and encrypted in the back end.
- Permissions: use access control lists (ACLs) to determine who (user and/or processes) can access data and what they can do with it. Another desirable layer of security are enterprise virtual private networks that allows to restrict accesses to employees via Two-Factor Authentication.
- Encryption: obscuring every tenant's critical data so that it will remain inaccessible to unauthorized entities even with data leaks.

Certain security patterns can be used in all the possible scenarios (Separate Databases, Shared Database with Separate Schemas and Shared Database with Shared Schema)

- Trusted Database Connections
- Secure Database Tables
- Tenant Data Encryption

2.1 Trusted Database Connections

Impersonation is one method to secure access to data stored in databases (Figure 3). In this approach, the database is configured to allow individual users to access different tables, views, queries, stored procedures, and other database objects. Any action that requires a call to the database use the user's security context to prevented undesired behaviour. It's possible to create a database user for each tenant, restrict its access only to strictly required stuff and adapt the application to connect to the database on behalf of each user.

Another approach is the trusted subsystem access method (Figure 4). In this scenario the application always connects to the database using its own application process identity. Any additional security permission layer must be implemented within the application. This approach is easier to handle, but it would not allow deeper permissions granularity.

2.2 Secure Database Tables

It's possible to secure a database with table granularity using the SQL GRANT command. The following command will add the user account to the access control list for the table, allowing to restrict access only to a specific tenant user account:

```
GRANT SELECT, UPDATE, INSERT, DELETE ON [TableName] FOR [UserName]
```

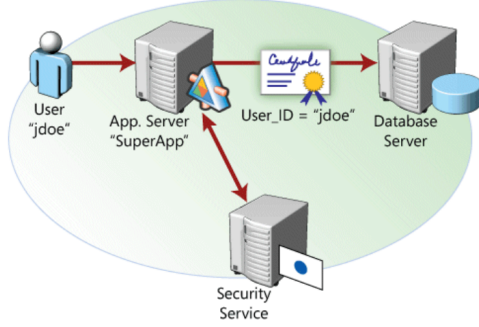



Figure 3: Impersonation database access

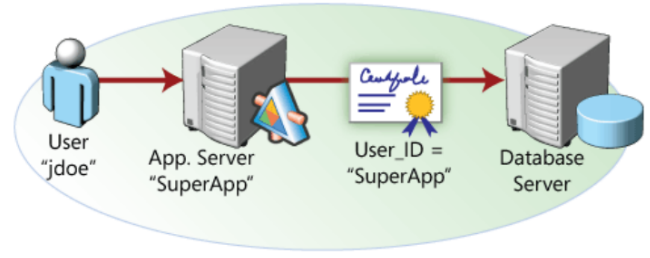


Figure 4: Trusted subsystem database access

It's possible to restrict tenant access to their rows creating views, adding an extra layer of data isolation in shared schema approaches. A view is a virtual table defined by the results of a SELECT query. If there is a table called *Table* and several tenants {Tenant1, ..., TenantN} it's possible to prevent direct read access to *Table* and create several views {Tenant1TableView, ..., TenantNTableView}. Each TenantXTableView will be the result of a SELECT query filtering with the tenant id column.

```
CREATE VIEW Tenant1TableView AS
SELECT * FROM Table WHERE TenantID = SUSER_SID()
```

This statement obtains the security identifier (SID) of the tenant user account accessing the database. In general, the tenant id is different to the tenant's SID so an extra table linking them could be needed (directly using the tenant id as filter would be less secure). It's important to highlight that only read access could be restricted using this approach because the tenants still need to insert and update rows in the original table *Table*.

Malicious/mistaken UPDATE, INSERT, DELETE statements from unauthorized tenant accounts could also be prevented in shared tables. It could be done using a similar approach with SID creating triggers (shown in Figure 5)

```
DELIMITER $$

CREATE TRIGGER
    TENANT_PREVENT_INSERT
    BEFORE INSERT
    ON Table FOR EACH ROW
BEGIN
    IF NEW.TenantID !=
        SUSER_SID() THEN
        -- Throw error
    END IF;
END$$

DELIMITER ;
```

```
DELIMITER $$

CREATE TRIGGER
    TENANT_PREVENT_UPDATE
    BEFORE UPDATE
    ON Table FOR EACH ROW
BEGIN
    IF OLD.TenantID !=
        SUSER_SID() THEN
        -- Throw error
    END IF;
END$$

DELIMITER ;
```

```
DELIMITER $$

CREATE TRIGGER
    TENANT_PREVENT_DELETE
    BEFORE DELETE
    ON Table FOR EACH ROW
BEGIN
    IF OLD.TenantID !=
        SUSER_SID() THEN
        -- Throw error
    END IF;
END$$

DELIMITER ;
```

Figure 5: MySQL base trigger examples to prevent insert/update/delete different tenant data

2.3 Tenant Data Encryption

Another security layer is database data encryption. Cryptography can be categorized as either

- Symmetric: a key is generated to encrypt and decrypt data.
- Asymmetric (public-key cryptography): two keys are generated, the public key is used to encrypt and the private key to decrypt.

Public-key cryptography requires significantly more computing power than symmetric cryptography. Encryption is a well-known technique, in multitenancy is possible to add an extra layer of security by creating different keys for each tenant.

3 Extensibility

Extensibility measures the ability and effort required to expand a system's behaviour. It's a convenient pattern in the case of evolving systems which can not be designed entirely in advance. Extensions of system's functionality increase the complexity and could make software maintainability harder. Just because it's possible to extend the software functionality doesn't mean we should. An extensible design provides a framework to allow changes [5], small increments will improve software reusability.

Traditional software development requirements will be needed for appropriate extensibility [5]:

- High cohesion: degree to which part of the code forms a logically single and atomic unit. Example: classes that contain strongly related functionalities help to keep related parts of a codebase in a single place.
- Low coupling: degree of interdependence between software modules. Changes in more independent modules will not heavily impact other modules.
- Interface-implementation separation: allows you to change the implementation independently of the interface. The use of polymorphism allow several implementations of the same interface in order to do similar things in different ways.
- Dependencies: external dependencies are often needed to perform specific tasks. When a system grows is usually divided into several components, there will be internal inter-dependencies.

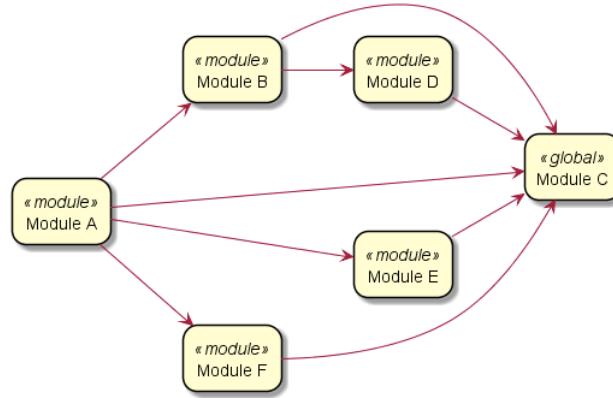


Figure 6: Dependency graph

It's possible to automatically generate the dependency graph in order to inspect coupling or dangling dependencies (Figure 3). External dependencies tracking allows for the recognition of possible migrations to avoid non-maintained or vulnerable dependencies. Extensibility would lead to fewer and cleaner dependencies and well-defined interfaces [5].

- Procedures to perform continuous integration: automate the integration of code changes from multiple contributors into a software project. Usage of automated tools for build, test and code analysis will allow faster feedback on new code's correctness [7].

Continuous Integration (CI) is generally used alongside an agile software development workflow [7]. The next step is Continuous Delivery, responsible for packaging an artifact together ensuring that software is always ready to be delivered [7]. Continuous deployment is the final phase of the pipeline. The deployment phase is responsible for automatically launching and distributing the software artifact to end-users, automatically pushing code out when tests pass [7].

The extensible design fits well with Agile methodologies and iterative development, allowing re-prioritisation [5] looking at software as a growing entity.

3.1 Multitenancy customization

It's possible to conditionally extend a tenant functionality by addition of new features or through modification of existing ones. There will be a set of tables/columns at the core of the application, but custom customer needs are not always addressable with a rigid data model.

There are three main ways to extend the data model to handle custom functionality [1]:

- Preallocated Fields
- Name-Value Pairs
- Custom columns

The data model can be customized creating custom fields in certain table(s) to allow tenants to extend their functionality [1] (Figure 7). This black box approach allows to determine how to use these fields depending on each customer, hiding actual data model complexity. Nullable strings could be used as a flexible data type for the custom fields, avoiding unnecessarily data type restrictions (castings can be done in the code).

TenantID	FirstName	BirthDate	C1	C2	C3
345	Ted	1970-07-02	null	"Paid"	null
777	Kay	1956-09-25	"66046"	null	null
1017	Mary	1962-12-21	null	null	null
345	Ned	1940-03-08	null	"Paid"	null
438	Pat	1952-11-04	null	"San Francisco"	"Yes"

Figure 7: Database table with preallocated fields C1, C2 and C3

Too few custom fields restrict tenants' customizability. Too many custom fields will lead to a sparse database with many unused fields. Name-Value pairs allow each tenant to define an arbitrary number of custom fields [1] using one to many relationships with an external table (Figure 8). When the application retrieves a record from *original_table*, it performs a lookup in the extension table *name_value_table*, selects all name-value pairs corresponding to the record id. It's a good approach when specific customers require considerable flexibility and the shared database approach is used. The disadvantages of name-value approach are:

- Extra complexity for database functions (such as indexing, querying, and updating records)
- Black box approach with hidden types and possible database relationships
- Possible anti-pattern in the code

```
if isTenant1():
...
elif isTenant2():
...
```

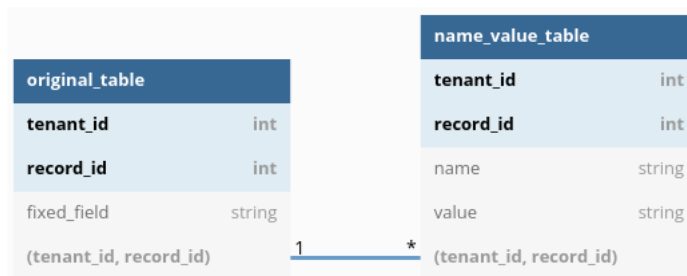
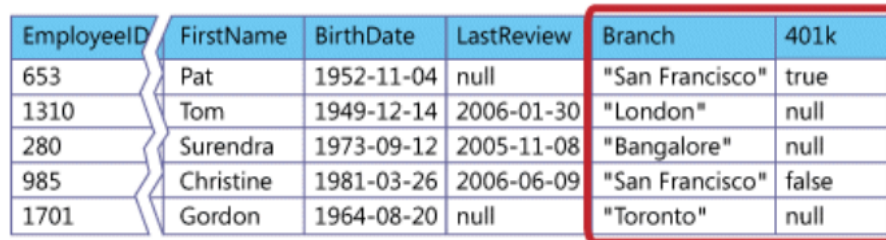


Figure 8: Tenants custom fields using name-value pairs

Another extensible data model pattern is adding tailored columns to tenants' tables (Figure 9). This pattern is appropriate mostly for separate-database or separate-schema applications. There would be extra unneeded fields for several customers in a shared database environment when different tenants have radically different requirements.



EmployeeID	FirstName	BirthDate	LastReview	Branch	401k
653	Pat	1952-11-04	null	"San Francisco"	true
1310	Tom	1949-12-14	2006-01-30	"London"	null
280	Surendra	1973-09-12	2005-11-08	"Bangalore"	null
985	Christine	1981-03-26	2006-06-09	"San Francisco"	false
1701	Gordon	1964-08-20	null	"Toronto"	null

Figure 9: Custom database columns extensibility approach

Regardless of the extensibility pattern(s), any custom field will require a modification to the business logic, the presentation logic, or both.

3.2 SaaS architecture approaches

3.3 Deployments

3.4 Software regressions

A software regression is a special type of bug where a certain feature was working before and stops working. Regressions can be produced due to changes in the code or the environment (i.e. system upgrades, environment variables, ...). Regressions can be classified in:

- Functional: a feature that is no longer working
- Performance: a feature is still working but more slowly or uses more resources (i.e. memory) than before

Testing can reduce regressions likelihood, and tests are either manual (sometimes involving Quality Assurance positions) or automatic.

3.4.1 Automatic testing

Automatic tests will catch bugs before production release and several strategies that can be used together:

- End to end tests (E2E): involves entire workflows, typically front end, back end and database(s).
- Integration tests: specific use cases involving several components of a system.
- Unit tests: test cases involving only one component, mocking other components when needed. For example: in the back end, the database calls are not involved (just mocked), making the tests much faster.

The most convenient relationships between the number of tests for each strategy are described in the test pyramid:

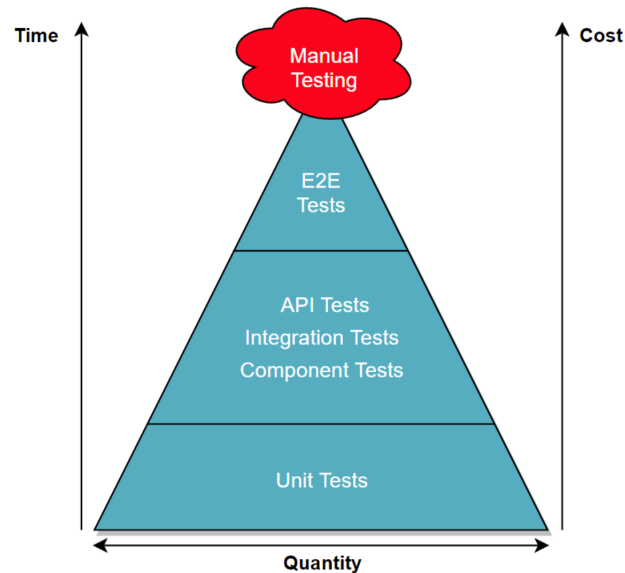


Figure 10: Test pyramid [2]

The inverted testing pyramid (fewer unit and integration tests, with an emphasis on automated and manual functional testing) is considered an anti-pattern, reducing the responsiveness, maintainability and reliability of the test setup [2]. Unit tests help identify faults in earlier phases [2] and must be the backbone of any software product.

Regression testing validates that the modified parts of a program do not introduce unexpected errors [6]. Unit tests can be introduced in the continuous integration loop because their feedback is almost instantly End to end and integration tests do not scale well so it's not always possible to include all of them in the continuous integration loop. Regression test selection techniques reduce the cost by selecting only the test cases related to a modified program (i.e. two consecutive commits in a version control system) [8].

There are other types of tests that should be considered:

- Performance tests: check the system when it is under significant load, they are quite costly to implement and run.
- Smoke tests: check basic functionality of the application.
- Security Testing: check how the system reacts to internal and/or external threats (i.e. penetration testing).
- Mutation tests: the code of a program is changed and verifies whether the existing test cases can identify these defects. It's a way to tests your tests, weak tests can produce a false sense of safety.

3.4.2 Tests detection

3.5 Observability and monitoring

Observability has its roots in control theory, addressing how to infer the internal state of a system by looking at its output.

4 Performance

Large-scale enterprise software is meant to be used by thousands of people simultaneously. Performance is relevant in multi-tenant applications where a bottleneck affecting one customer could potentially affect several customers. We need to distinguish between data scaling (increased capacity for storing and handling data) and application scaling (total workload that the application can handle) [1].

4.1 Database

Databases can be scaled up increasing server resources (more powerful processors, more memory, quicker disk drives, ...) and scaled out (partitioning one database onto multiple servers) [1].

4.2 Data scalability patterns

There are two main tools to scale out a database:

- Replication: copy all or part of a database to another spot(s), and then keep synchronized the copy (or copies) with the original. If only the original database can be written to is called single-master replication. When it's possible to write into several copies of the database is called multi-master replication and requires a more complex synchronization mechanism to integrate the changes between the different copies. It's a good approach to create read-only copies of data that doesn't change very often.
- Partitioning: subsets of the database are moved to other locations/databases. In horizontal partitioning, a table is divided into several databases using the same structure, resulting in tables with fewer rows. In vertical partitioning, a table is divided into smaller tables with the same number of rows but splitting the columns from the original table. This technique is useful when a monolithic application is splitted into several modules. Splitting the tables in a schema into several subsets allows for independently deployable modules. As a rule of thumb, try to find divisions in the data minimizing the number of cross-database communications.

It's possible to create horizontal partitioning in a shared database based on tenant ids. Different tenants can have very different demands. It's necessary to prevent overtaxed partitions while other partitions are underused, i.e. partition the database to equalize the total number of active end users on each database.

4.2.1 Concurrent data accesses

4.3 Cache

4.4 Quality of Service in Cloud Computing

5 Petri nets

A Petri net is a directed graph with two kinds of nodes, interpreted as places and transitions, such that no arc connects two nodes of the same kind.

5.1 Concurrent systems validation

6 Use case: e-commerce platform

The system will store products (each product belonging to a specific tenant) and will allow normal users to order them. Each tenant could be an individual or organization and it will have a back office available to customize their product(s) and keep track of the orders.

Technological stack of the system:

- Front end: Angular (testing with Karma + Jasmine)
- Back end: Java 11 (testing with JUnit)
- Database: MySQL (with agnostic database calls)
- End to end tests: Selenium
- Code analysis: SonarQube

6.1 Coverability Trees using Petri Nets

6.1.1 Availability

6.1.2 Vivacity?

6.1.3 Security

References

- [1] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. “Multi-Tenant Data Architecture”. In: *Microsoft Corporation* (2006).
- [2] Ville Hartikainen. “Defining suitable testing levels, methods and practices for an agile web application project”. In: *LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT* (2020), pp. 18–19.
- [3] Jaap Kabbedijk et al. “Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective”. In: *The Journal of Systems and Software* (2014).
- [4] Seungseok Kang et al. “A General Maturity Model and Reference Architecture for SaaS Service”. In: vol. 5982. Apr. 2010, pp. 337–346. ISBN: 978-3-642-12097-8. DOI: 10.1007/978-3-642-12098-5_28.
- [5] Allan Kelly. “The Philosophy of Extensible Software”. In: *ACCU* (2002).
- [6] Amir Ngah, Malcolm Munro, and Mohammad Abdallah. “An Overview of Regression Testing”. In: *Journal of Telecommunication, Electronic and Computer Engineering* (2017).
- [7] Max Rehkopf. “What is continuous integration?” In: *Atlassian documentation* (2022).
- [8] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. “Regression test selection for C++ software”. In: *Journal of Software: Testing, Verification and Reliability* (2000), pp. 77–109.
- [9] Qi Zhang, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges”. In: *Journal of Internet Services and Applications* (2010), pp. 7–18.