УДК 004.89
DOI: 10.18523/2617-3808.2020.3.69-74

K. Gorokhovskyi, O. Zhylenko, O. Franchuk

DISTRIBUTED SYSTEMS TECHNICAL AUDIT

Modern enterprise systems consist of many deployment artifacts, thus, microservices architecture is extremely popular now. The use of distributed systems is rapidly growing despite the increased complexity and the difficulty of building. The main reason for such a trend is that the advantages of distributed systems outweigh their disadvantages. Nevertheless, product release into the market is not a final step of software development lifecycle. Next important step is maintenance that continues much longer than development. System failures and delays in finding and fixing a problem can cause huge financial and reputational expenses. In addition, the new features introduced due to changes on the market should take place on time.

Prior to releasing a product to market, we would like to know in advance possible technical gaps to understand what we can expect and maybe fix some issues in order to save time and money in future. In other words, we should be able to make a decision, if the product is ready for launch or not, relying on some data. Such analysis is as well necessary when we obtain ownership for an existing product. Technical audit helps to find out technical debt and assess risks related to maintenance and extension of the system. It should be considered as mandatory activity during release preparation and ownership transfer. Well-defined criteria would help to conduct an audit smoothly and find out most of the technical debt. Properly conducted technical audit reduces risks of problems after release but it does not guarantee commercial success of product or absence of problems at all.

In this article we will define what distributed systems are, we will review Monolithic, Microservices and Serverless architectures, describe what are quality attributes and what should be taken into account during technical audits. Next, we will deep dive into the technical audit process, specify what aspects of the system must be considered during an audit. Then we will iterate over checklists items in order to provide guidelines based on the best practices in industry which helps to prepare for software system audit.

Keywords: Distributed systems, Monolithic, Microservices, Serverless, Quality attributes, Observability, Portability, Security, Maintainability, Technical audit, Checklist.

INTRODUCTION

It is really difficult to imagine enterprise system that consist of only one deployment artifact. Great example is microservices architecture which is extremely popular now. But when we want to release product to market or we receive existing product on ownership, we need to know technical gaps in advance to understand what we can expect and maybe fix some issues upfront in order to save time and money in future.

In this work we defined what technical audit is and what aspects of distributed systems we need to consider. Also, we provided checklists that are based on best practices in IT industry that can help to conduct audit smoothly.

Distributed systems overview

Initially applications rely on persistent connections and stateful communication. All heavy processing was on backend part. The frontends were thick however we had rich user interface, complex business logic and data access. Separation of concern was used to manage complexity. There was three common layers: presentation, business and data access. But dependencies between them became so complex so it was real challenge to introduce new functionality and still all the functions are managed and served in one place. Monolithic applications have lack modularity because it is one large code base. If something even small must be updated or changed developer access the same code base and make changes in the whole stack at once.

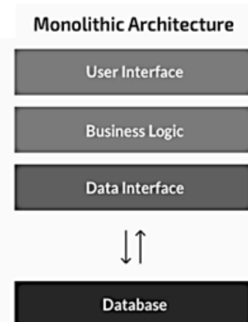

Figure 1. Layers of monolithic architecture

Microservices architecture breaks single unit into a collection of smaller ones which are not depend on each other. These units are considered as separate services each of them has its own logic, storage and they concern on specific functions.

© *K. Gorokhovskyi, O. Zhylenko, O. Franchuk,* 2020

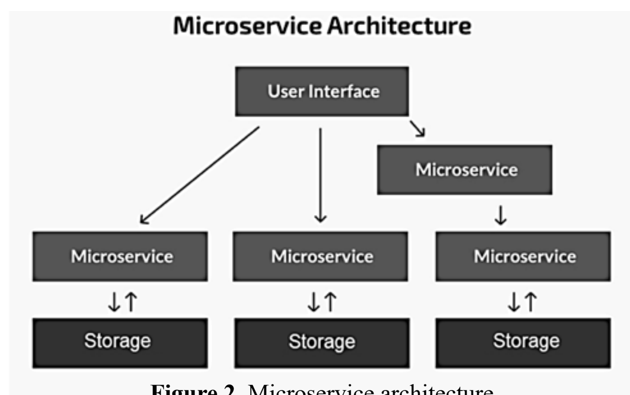

Figure 2. Microservice architecture

Serverless is a cloud computing execution model where the cloud provider dynamically manages the allocation and provisioning of servers. A serverless application runs in stateless compute containers that are event-triggered, ephemeral (may last for one invocation), and fully managed by the cloud provider [3]. Pricing is based on the number of executions.

Considering different architectures, we see that main difference in monolithic and distributed architecture is possibility to release different functionality independently. Microservices and serverless are not panacea but nowadays distributed systems are most common style for enterprise systems. You must take into account that together with granularity of your system you increase system complexity on supporting infrastructure.

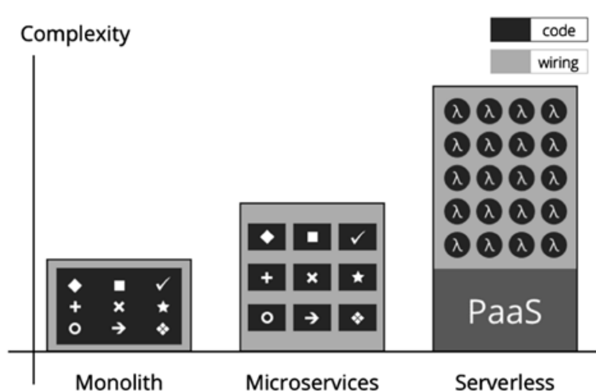

Figure 3. Difference between monolithic, microservice and serverless architectures [7]

Technical audit overview

Audit is a formal procedure to measure a technical debt and a quality level of the system. The main purpose of audit procedure is checking compliance of a software system (or a software component) and an infrastructure with well-known and up-to-date practices in industry.

Following types of a technical debt are covered by technical audit:

- **Production technical debt.** It nest high risks for a product. It focuses on observability (e.g., effort to find and fix an issue), portability (e.g., effort to release a new version) and security (e.g., any security vulnerabilities).
- **Development technical debt.** It nest moderate risks for a product. It focuses on maintainability (e.g., effort to introduce changes into existing system).
- **Involvement technical debt.** It nest moderate risks for a product. It focuses on understandability (e.g., effort to introduce a new employee on project).

There are 2 general cases when you need to perform audit:

- **During release preparation.** Team want to know problems and potential issues that might be in production after release.
- **During ownership transfer.** When team take ownership on existing product, they need to know gaps that can lead to problems in production. Also knowing problems help to provide accurate estimation on new feature introduction.

There are several main phases of audit:

- **Kickoff.** Define scope and schedule of audit.
- **Preparation.** Define and approve checklists.
- **Execution or examination.** Check system compliance according to prepared checklists.
- **Reporting.** Build report and include all findings.

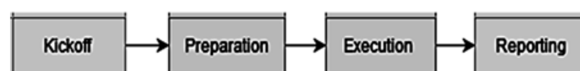

Figure 4. Phases of technical audit

You should take into account that main goal of audit is to identify technical debt. Investigation for possible fixes is not mandatory part of a system. After report is ready it should be reviewed and after that decision must be made what issues are critical and requires immediate actions.

Quality attributes

First of all, it is better to start with classification of requirements.

All requirements encompass the following categories:

- Functional requirements
- Non-functional requirements
  - Quality attributes
  - Constraints

Functional requirements define a system or its component. They describe what the system must perform and how to behave or react on stimulations at runtime. These requirements describe specific functionality that define what a system is supposed to do (they involve calculations, technical details, data manipulation and processing, etc.)

Quality attribute requirements are qualifications of functional requirements or of the overall product. They usually answer questions like 'how fast the function should be performed' or 'how resilient it should be to incorrect input' can be considered as qualifications of functional requirement. The overall product qualifications are such items as 'time to deploy the product' or 'how fast new feature must be introduced'.

Constraint is a design decision taken with zero degree of freedom. This decision that has already been taken and we cannot change it. The examples of constraints are decisions to use a particular language or reuse certain module, or management directive to use specific cloud provider like AWS. Such decisions usually based on some external factors like company has long term investment in AWS.

Observability

Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. Usually developers confuse monitoring and observability. Observability is a property of a system in contrast to monitoring whereas monitoring refers to the process where we translate application and infrastructure logs and metrics data in order to be able to provide meaningful actions.

If system and its components don't adequately externalize their state, then even the best monitoring can fail.

Portability

Portability is the ability to deploy a product in various environments in a predictable way. It includes containerization, configuration and versioning. Docker is default tool for containerization. Configuration and versioning are implemented by custom solution and may be various by standards in different organizations.

Security

Security is the ability to resist to incorrect or malicious behavior of client applications.

Here is the list of the main security areas:

1. Authentication and authorization of clients.
2. Translation, interpretation and protection of data.
3. Configuration management and dependency management.
4. Monitoring, logging and auditing.

There several projects which provide list of top vulnerabilities. OWASP [5] and CWE [1] are most popular. These lists should be considered during system development. OWASP Top 10 is the most popular list which represents a broad consensus about the most critical security risks to web applications.

Maintainability

Maintainability is the ability to change a product with a predictable effort. Static analysis is common approach that used to control maintainability. However, these tools may not provide enough checks to ensure maintainability, so code review practice is recommended also.

The major recommendations are:

Minimize source code

For example, in Java Lombok library can be used auto-generated getters, setters and constructors to enable dependency injection (Spring Framework).

Prefer declarative configurations

For example, use declarative clients instead of request builders to consume data from HTTP services.

Prefer infrastructure solutions

For example, configure a reverse proxy instead to enable CORS. Do not use an application framework for this purpose.

Technical audit checklists

Observability checklist

Use correlations. Unique correlation identifier must be assigned to each invocation. This identifier must be propagated to all services, message brokers and event buses. It used to identify side effects in system for single user intent.

Enable monitoring. Metrics must be collected and aggregated in single place. Most important metrics are latency, throughput, errors and utilization. By default, next tools are used: Prometheus – to store metrics, Grafana – to visualize metric.

Enable logging. Collect logs from every part of a system in single place. All requests from external services, message brokers, event buses and data stores must be traced. Sensitive information must be masked. By default, next tools are used: Fluentd – to aggregate logs, Elasticsearch – to store logs, Kibana – to visualize logs.

Use log context for instances. It is useful to log application version and configuration properties except secrets on startup. It reduces time on troubleshooting.

Enable error handling. Provide a default error handler. Next fields must be included into the error response for all error handlers: application name, instance identifier, correlation identifier and error code.

Use health checks. Each service should have endpoint which provide information about service health status. By default, HTTP method GET is used which returns HTTP status 200.

Enable tracing. Trace must be propagated to all services, message brokers and event buses. This information must be collected on aggregation tool.

Log context for invocations. Log a security and operation identifiers for each invocation. These identifiers allow to extract the contextual information about client and operational behavior.

Enable error tracking. All errors should be tracked. Logs also contains error but it is essential to have additional separate place for errors that provides possibility to send notification if necessary.

Portability checklist

Enable containerization. Images should be used to deliver components. Configure repository and version for images.

Use immutable tags. Use immutable tags. Avoid to use latest tag.

Follow to best practices for images. Use docker best practices to build images since Docker is default tool for containerization.

Use external configuration. Do not embed configuration inside image. Configuration file must be separate from image. There should be possibility to override configuration properties via environment variables.

Use versioning. Versioning must be used for images and also recommended for configurations files.

Don't embed infrastructure into services. SSL termination, rate limiting, CORS and so on should be configured using infrastructure not application.

Define quotas for CPU and memory. Leaks in resources for single service should not crush machine on which other service might run. Also, it helps orchestrator to find appropriate node in cluster faster.

Security checklist

Segregate services by security traits. Services should be segregated by access type (public, internal and so on) and by required privileges. PoLP principle should be used (principle of least privilege).

Validate inbound data. Validate all incoming requests, responses, messages and events before start processing them.

Don't expose sensitive data. Exposed sensitive data may be used by attackers to compromised this data also this may lead to fines from regulators. Do not show stack trace to client. This can be used to enable negative impact on system.

Control dependencies versions. Regularly update dependencies because updates nest fixes for vulnerabilities.

Maintainability checklist

Use branching strategy. It helps developers work separately and do not affect each other. All changed in main branch should be done through pull requests.

There are three primary strategies that is widely used: GitHub Flow, GitLab Flow, Git Flow. More information can be found in [6].

Enable build automation. Use single build script for local developers' machine and remote automation builds. All infrastructure tasks such as static code analysis should be removed form build script.

Use unit tests. Test coverage may be different depends on organization standards but recommended coverage is higher than 60 %.

Define feedback activities. Define all activities and quality gates that must be passed before code will be transferred to next stage. It helps shift feedback to the left of feedback activity diagram and find out problems earlier.

The cost of detecting and fixing defects in software increases exponentially with time in the software development workflow. Fixing bugs in the field is incredibly costly, and risky – often by an order of magnitude or two. The cost is in not just in the form of time and resources wasted in the present, but also in form of lost opportunities of in the future [2]. For example, it is much harder fix bug that was found in production than during execution of unit tests and even during execution of automation e2e tests.

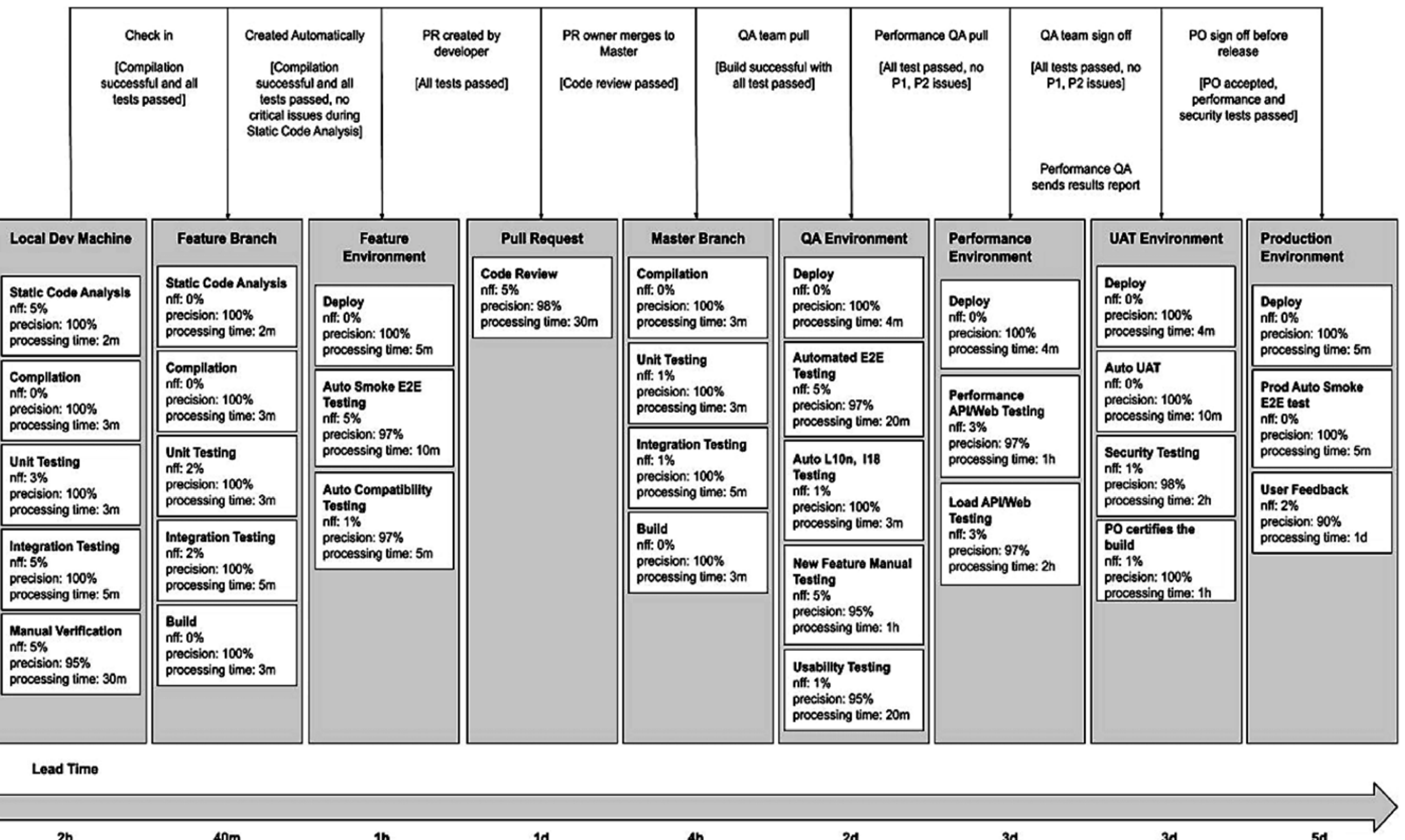

Figure 5. Feedback activities diagram example

Use code conventions. It improves readability of the code. It is recommended to have single convention in scope of all system but at least it should exist in scope of single team. Code convention template can be exported into file and shared with team. All modern IDEs support such feature.

Reduce code duplication. It is recommended to have less than 3 % of code duplication. Static code analysis cannot find semantic code duplication that is why code review is necessary.

Remove dead code. Remove unused or commented code. Previous implementation can be reverted from git log history.

Ensure methods and classes maintainability. Use clean code principles. These principles were described in a book Clean Code by Robert C. Martyn [4].

Summary

Maintenance of the system after release is very important part of software life cycle. System failures and delays in finding and fixing a problem can cause to huge financial and reputational expenses. Also, the introducing of new features due to changes on the market should take place on time.

Technical audit helps to find out technical dept and asses risks due to maintenance and extension of the system. It is a mandatory during release preparation and ownership transfer. Well defined criteria will help to conduct audit smoothly and find out most of technical dept. It does not guaranty success of product or absence of problems but properly conducted technical audit reduce risk to have them after release.

References

1. Common Weakness Enumeration. (2019). *2019 CWE Top 25 Most Dangerous Software Errors.* Retrieved from https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.
2. DeepSource Corp. (2019). *Exponential cost of fixing bugs. How the cost of finding and fixing defects increases with time.* Retrieved from: https://deepsource.io/blog/exponential-cost-of-fixing-bugs.
3. Gnatyk, Romana. (2018). *Microservices vs Monolith: which architecture is the best choice?* Retrieved from https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business.
4. Martyn, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
5. OWASP. (2020). *Top Ten Web Application Security Risks.* Retrieved from https://owasp.org/www-project-top-tenjf.
6. Porto, Patrick. (2018). *4 branching workflows for Git.* Retrieved from https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf.
7. Solanki, Jignesh. (2017). *Evolution of Serverless: Monolithic Microservices FaaS.* Retrieved from https://dev.to/jignesh_simform/evolution-of-serverless-monolithic-microservices-faas-3hdp.

Гороховський К. С., Жиленко О. В., Франчук О. В.

ТЕХНІЧНИЙ АУДИТ РОЗПОДІЛЕНИХ СИСТЕМ

У статті наведено визначення розподілених систем і розглянуто Monolithic, Microservice та Serverless архітектури. Описано процес технічного аудиту та уточнено аспекти системи, які потрібно враховувати під час аудиту. Розглянуто атрибути якості. Наведено контрольні списки для аудиту, основані на найкращих практиках у галузі, що допомагає підготуватися до технічного аудиту.

Ключові слова: розподілена система, Monolithic, Microservice, Serverless, Quality атрибут, спостережливість, портативність, безпека, ремонтопридатність, аудит, контрольний список.

Матеріал надійшов 10.06.2020

Creative Commons Attribution 4.0 International License (CC BY 4.0)