Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"

Network Technologies Department of the Faculty of Informatics

Distributed system technical audit

Text part of course work
in specialty “Computer Science and Information Technologies” 112

Coursework supervisor
associate professor, doctor of technical science
Hlybovets A.M.

_____
(signature)

“____” _____ 2020 yr.

Made by student
Zhylenko O.V.

“____” _____ 2020 yr.

Kyiv 2020

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"

Department of Computer Science of the Faculty of Informatics

APPROVED

Head of the Department of Informatics

PhD, associate professor

_____ S.S. Gorokhovsky

(signature)

"____" _____ 2020 yr.

INDIVIDUAL TASK
for course work

For the student of the Faculty of Informatics of 1 course of studying
THEME Distributed system technical audit

Output data:

Text part content of coursework:
An individual task
Calendar plan
Annotation
An introduction
Part 1: Distributed systems overview
Part 2: Technical audit overview
Part 3: Technical audit checklists
Summary
References
Applications

Issue date "____" _____ 2020 yr. Supervisor _____
(signature)

Task received _____
(signature)

Theme: Distributed system technical audit

Calendar plan of coursework execution:

№	Stage name	Deadline	Note
1.	Getting of coursework topic	30.01.2020	
2.	Searching of appropriate literature	05.02.2020	
3.	Reviewing materials and building the structure of coursework	10.02.2020	
4.	Reviewing distributed systems and writing first part	10.04.2020	
5.	Researching in technical audit practices	23.04.2020	
6.	Writing second part	24.04.2020	
7.	Investigation of quality attributes of distributed systems	30.04.2020	
8.	Writing third part	01.05.2020	
9.	Writing coursework summary	01.05.2020	
10.	Coursework analysis with the supervisor	02.05.2020	
11.	Coursework changing according to the supervisor’s remarks	03.05.2020	
12.	Creating of the presentation	05.05.2020	
13.	Defending of the coursework	14.05.2020	

Student Zhylenko O.V.

Supervisor Hlybovets A.M.

“____” _____

Contents

Annotation

In this coursework will be defined what is distributed systems, review Monolithic, Microservice and serverless architecture. Also, we will deep dive into technical audit process, specify what aspects of system must be considered during audit. Then will iterate over checklists item in order to provide guidelines based on best practices in industry that helps to prepare for system audit.

Key words: Distributed system, Monolithic, Microservice, Serverless, Quality attribute, Observability, Portability, Security, Maintainability, 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 promote product to production or we receive existing product on ownership we want 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 I will define what technical audit is and what aspects of distributed systems we need to consider. Also, I will provide checklists that are based on best practices in IT industry that can help to conduct audit smoothly.

1. Distributed systems overview

1.2 Monolithic architecture

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.

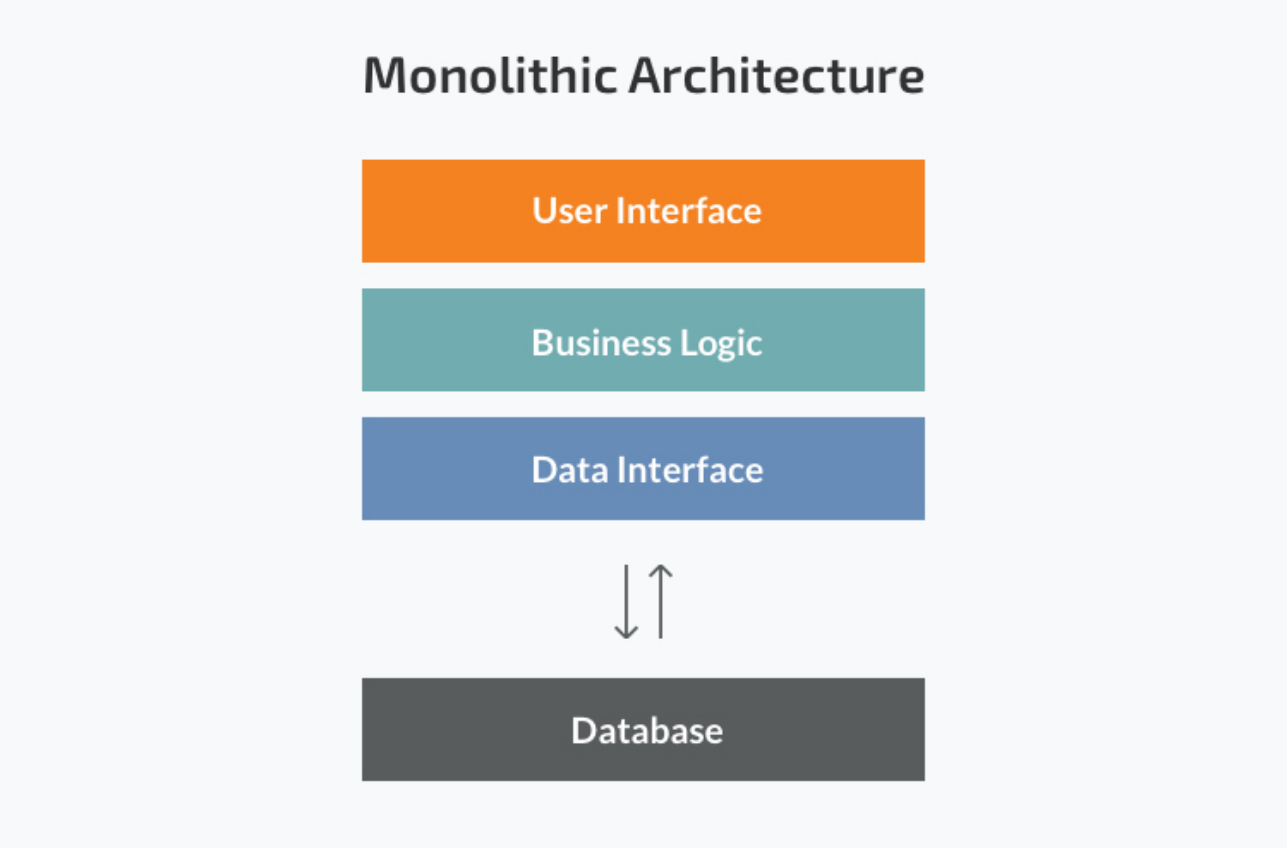

Figure 1.2.1. Layers of monolithic architecture

Advantages of monolithic architecture:

- **Less cross-cutting concerns.** These concerns affect the whole application and includes such staff like logging, monitoring, caching, etc. It is easier to handle because we have one application.
- **Easier debugging and testing.** In contrast to the other architectures.
- **Easier to deploy.** Need to deploy single unit.

At same time it has next disadvantages:

- **Understability**. Application tend to become really huge and difficult for understanding.
- **Scalability.** Impossible to scale components independently
- **Updatability.** Difficult to introduce changes.
- **Introduce new technologies.** Every new change affects the whole application.

1.3 Microservices 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.

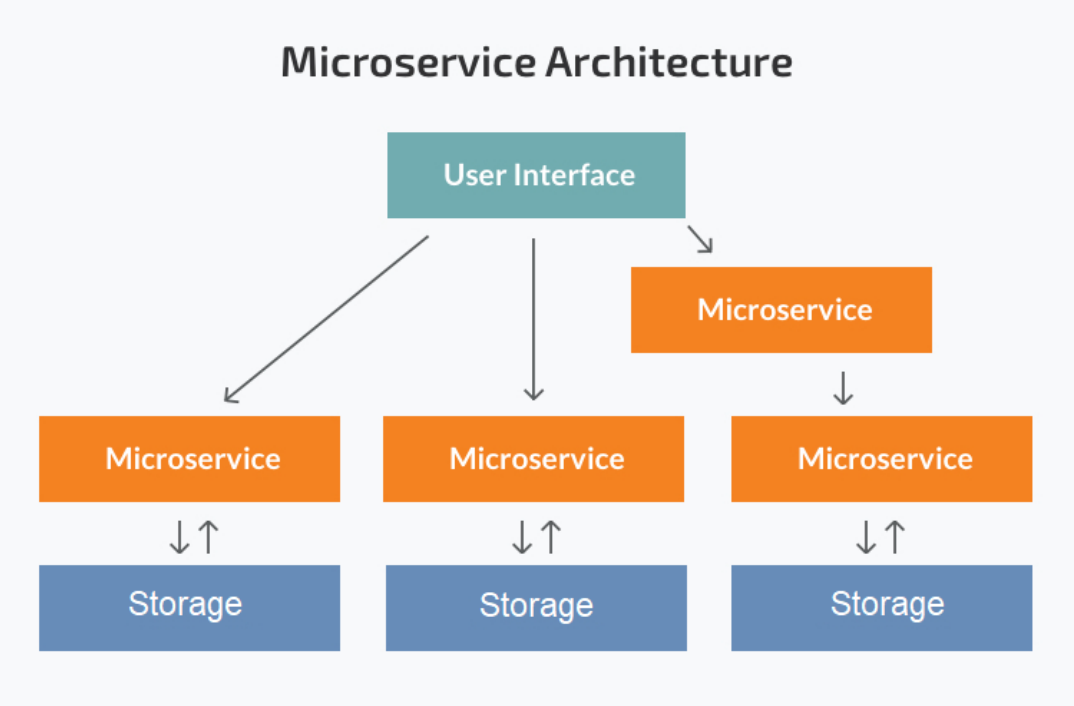

Figure 1.3.1. Microservice architecture

Advantages of microservice architecture:

- **Independent deployment units**. Services can be deployed independently.
- **Better understability.** Since each service is small and independent unit it is easier for developers to understand it.
- **Scalability.** Services can be scaled independently based on needs.
- **Agility.** One failing service do not crash the whole system. Other functionality still available.
- **Flexibility in choosing the technology.** Different frameworks and even languages can be used for different microservices.

But microservices have a list of disadvantages also:

- **Complexity.** since All these components must be connected between each other we move complexity from application to infrastructure layer.
- **Cross-cutting concerns.** We must take care about externalized configuration, logging, metrics, health checks, and others.
- **Testability.** Collection of components which are deployed independently makes testing much harder.

1.4 Serverless 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 [12]. Pricing is based on the number of executions.

Figure 1.4.1. Serverless in difference cloud providers [13]

Advantages of serverless architecture:

- **Focus on business logic**. We do not care about servers, load balancers and all such staff.
- **Scalability.** Virtually scalability is infinitive and managed buy cloud provider.
- **Pay for invocation.** We pay for execution of business logic.
- **Flexibility in choosing the technology.** As for microservice architecture we can use different frameworks and languages for different functions.

Disadvantages:

- **Unrelated set of functions**. Each function is just small piece of code and all functions must be connected to each other to perform real business function.
- **Testability.** Harder to test comparing with microservices.

1.5Comparing monolithic, microservice and serverless architectures

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.

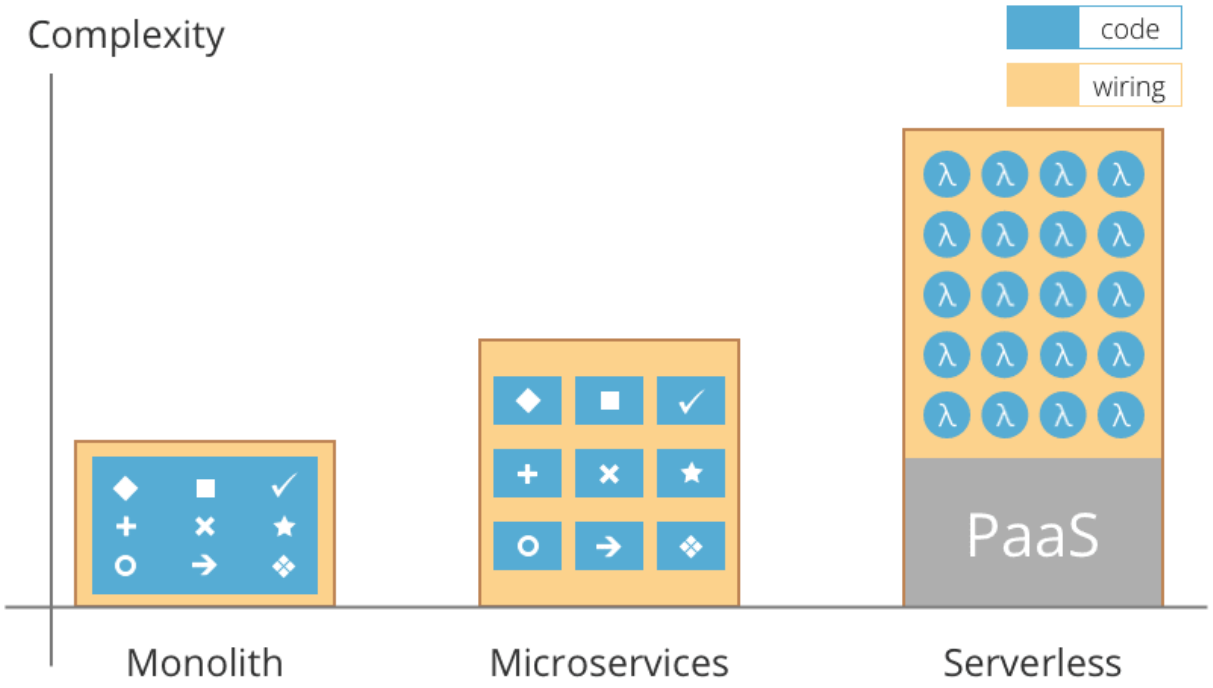

Figure 1.5.1. Difference between monolithic, microservice and serverless architectures [14]

2. Technical audit overview

2.1 Definition and purposes of technical audit

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:

- **Kikoff.** Define scope 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.

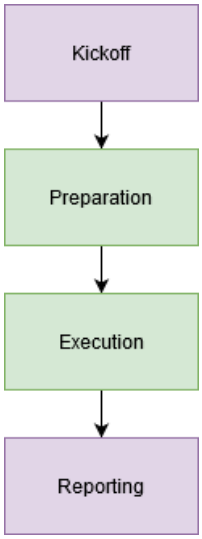

Figure 2.1.1 Phases of technical audit

You should take into account that main goal of audit is to identify technical dept. 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.

2.2 Quality attributes

2.2.1 Quality attribute definition

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’. We can used next definition proposed by SEI: “A quality attribute (QA) is a measurable or testable property of a system, that is used to indicate how well the system satisfies the needs of its stakeholders”

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.

2.2.2 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.

2.2.3 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.

2.2.4 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 and CWE 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 (https://owasp.org/www-project-top-ten/).

2.2.5 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:

1. **Minimize source code**

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

2. **Prefer declarative configurations**

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

3. **Prefer infrastructure solutions**

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

3. Technical audit checklists

3.1 Observability checklist

Next items have significant impact on observability and are highly recommended for implementation:

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.

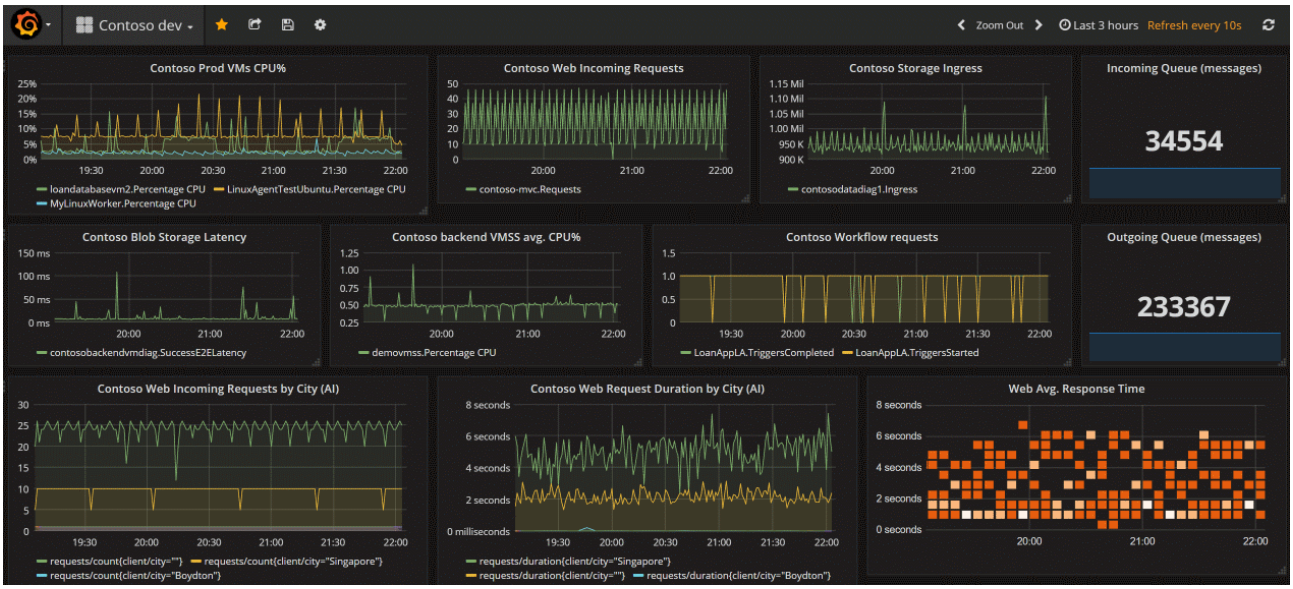

Figure 3.1.1. Grafana dashboard

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.

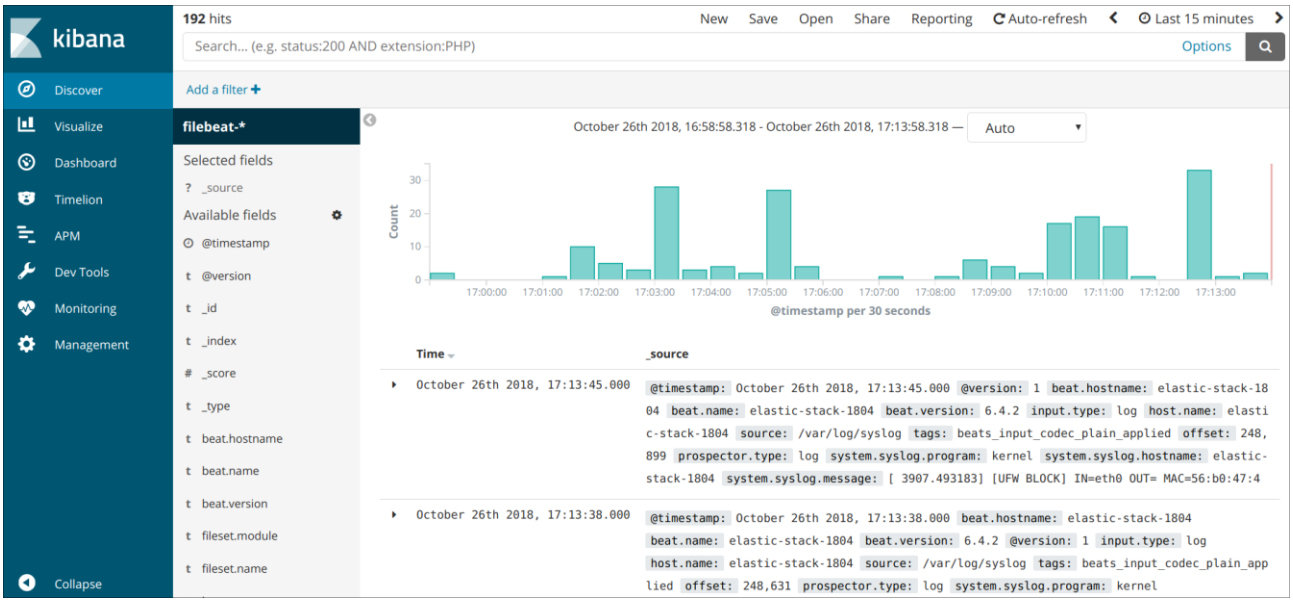

Figure 3.1.2. Kibana dashboard

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.

Following list of items related to should to have category:

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

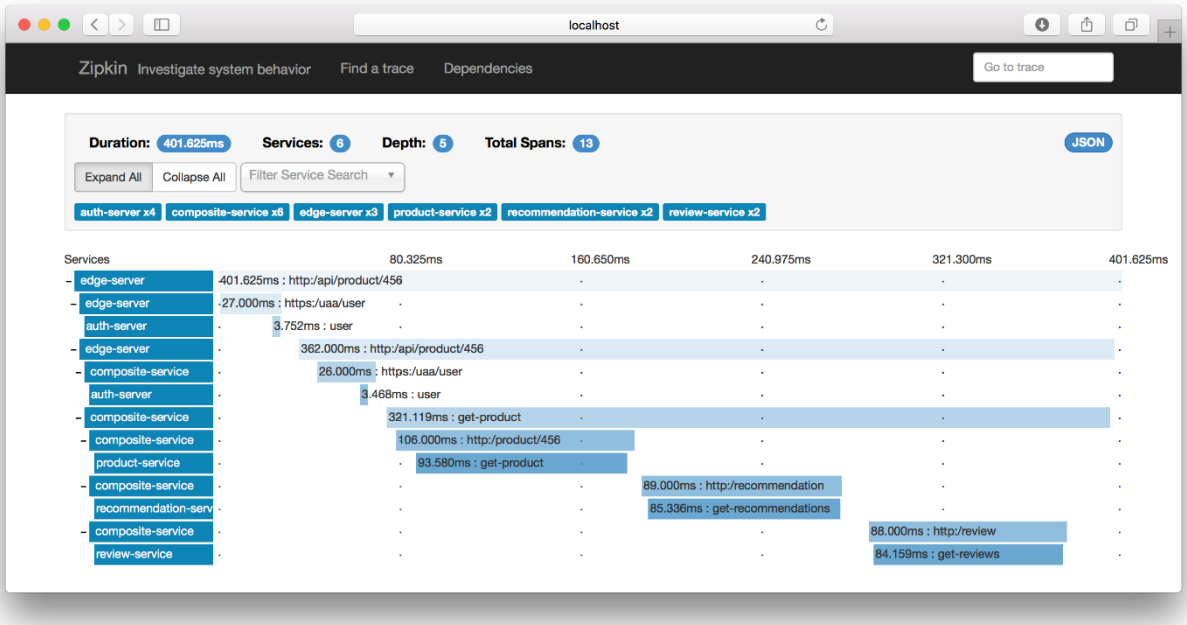

Figure 3.1.3. Zipkin tracing tool

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.

3.2 Portability checklist

Next items have significant impact on observability and are highly recommended for implementation:

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 conternization.

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.

Following list of items related to should to have category:

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.

3.3 Security checklist

Next items have significant impact on observability and are highly recommended for implementation:

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 privileges).

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.

3.4 Maintainability checklist

Next items have significant impact on observability and are highly recommended for implementation:

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.

GitHub flow is pretty simple and is best for small teams when team does not need to support several different versions or environments simultaneously. There is main *master* branch. When developer needs to introduce new changes, he creates feature branch. After implementation phase is completed developer creates pull request for review. When review is done and all comments were addressed *feature* branch can be merged into *master*.

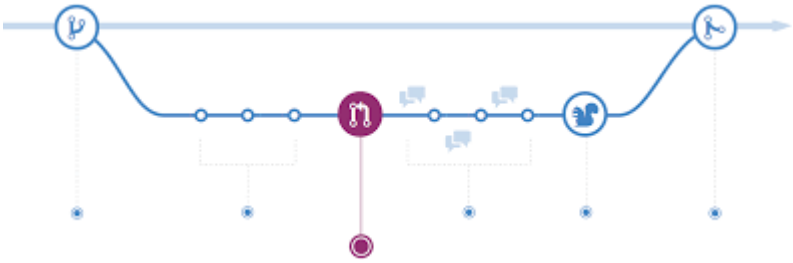

Figure 3.4.1. GitHub flow

GitLab Flow is good for the case where team have to support several different environments. There is still *master* branch and developer create *feature* branch from *master*. Additional branches are release-purposed for different environments. Team can continue work on new feature regardless of changes from the appropriate release branch but changes in *release* branch must be cherry-picked back into *master*. It is very useful if team need to deploy some changes to a pre-production environment without blocking other development activities.

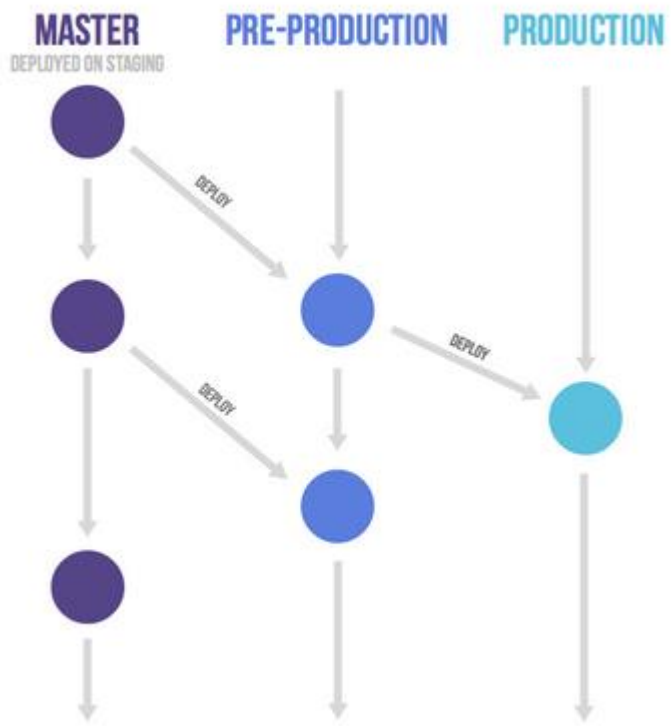

Figure 3.4.2. GitLab flow

Git flow was created by Vincent Driessen in 2010 and it is based in two main branches with infinite lifetime:

- *master* — branch with production code. All development code is merged into *master*.
- *develop* —branch contains pre-production code. All features are merged into *develop* when they are finished.

Several supporting branches are used:

- *feature-** — feature branches are used to develop new features for the upcoming releases. They are branched from *develop* and must be merged into *develop*.

- *hotfix*-* — these branches are used when we have undesired status of master and need to act immediately. They are branched from *master* and must be merged into *master* and *develop.*
- *release*-* — release branches support preparation of a new release in production. They used to fix many minor bugs and to prepare metadata for a release. They are branched from *master* and must be merged into *master* and *develop.*

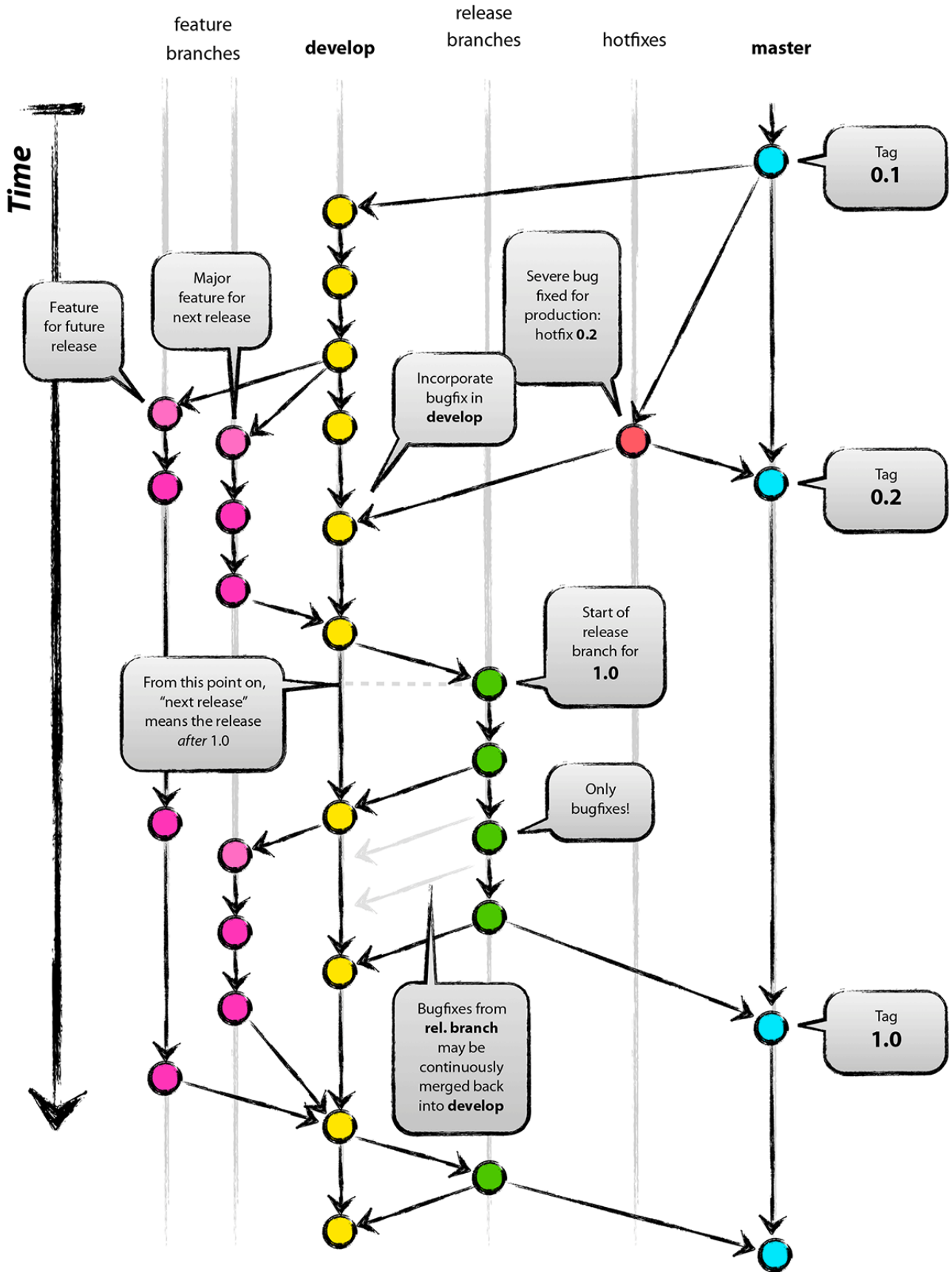

Figure 3.4.3. Git flow

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%.

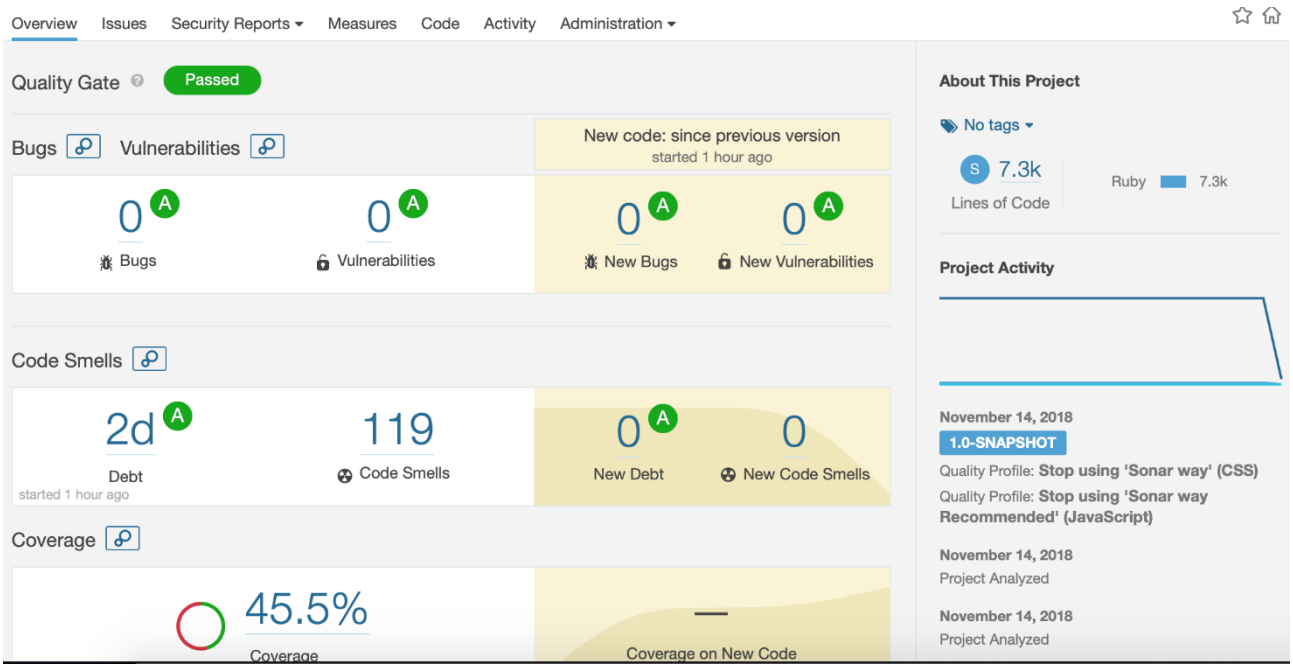

Figure 3.4.4. SonarQube dashboard

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.

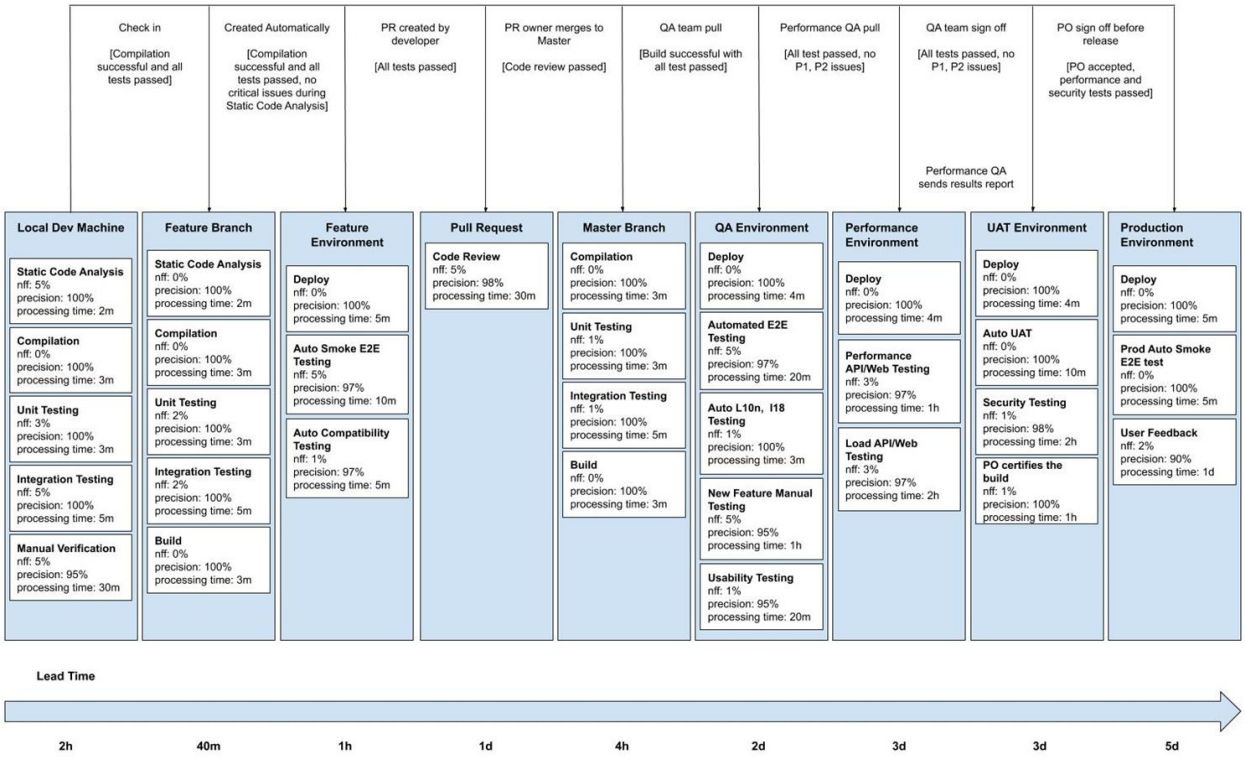

Figure 3.4.4. Feedback activities diagram example

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 [15]. 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.

Kent Beck in his book Extreme Programming Explained states that most defects end up costing more than it would have cost to prevent them. Defects are expensive when they occur, both the direct costs of fixing the defects and the indirect costs because of damaged relationships, lost business, and lost development time.

On following graph created by NIST (National Institute of Standards and Technology) we can see the effort in detecting and fixing defects increases through the phases of software development.

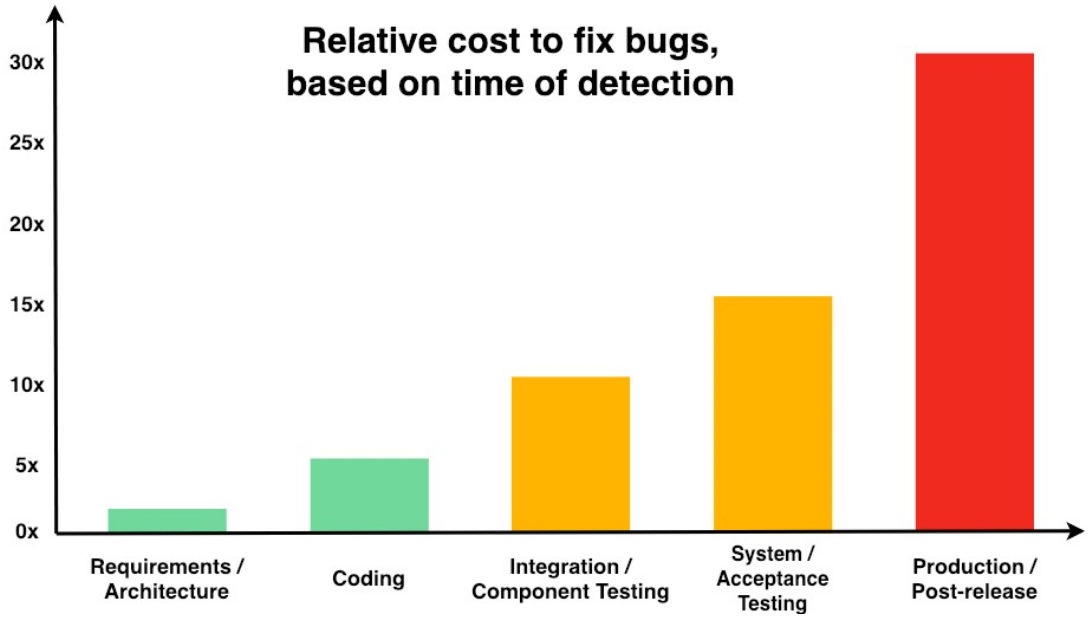

Figure 3.4.5. Relative cost to fix bugs based on time of detection [15]

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 book Robert C. Martyn, Clean Code: A Handbook of Agile Software Craftsmanship

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. https://thenewstack.io/monitoring-and-observability-whats-the-difference-and-why-does-it-matter/
2. https://www.bmc.com/blogs/observability-vs-monitoring/
3. https://owasp.org/www-project-top-ten/
4. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
5. Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy, *Site Reliability Engineering: How Google Runs Production Systems*
6. https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-tag-mutability.html
7. https://google.github.io/styleguide/javaguide.html
8. Robert C. Martyn, *Clean Code: A Handbook of Agile Software Craftsmanship*
9. https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf
10. https://jeffkreeftmeijer.com/git-flow/
11. https://medium.com/responsetap-engineering/monolith-to-microservices-to-serverless-our-journey-745a2b9620ec
12. https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
13. https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9
14. https://dev.to/jignesh_simform/evolution-of-serverless-monolithic-microservices-faas-3hdp
15. https://deepsource.io/blog/exponential-cost-of-fixing-bugs/
16. https://dzone.com/articles/go-microservices-part-12-distributed-tracing-with