Agile

Poor Scrum processes

FACTS

- No communication between PO and team
- Team members could be assigned to the several projects
- No retrospective
- No formalized scope management

IMPACT

- No opportunity for the Scrum team to inspect itself
- No opportunity to clarify requirements with PO

ADDITIONAL INFORMATION

Scrum is good framework by it’s constrains and backlog management. Current flow is more likely Kanban process.

RECOMMENDATIONS

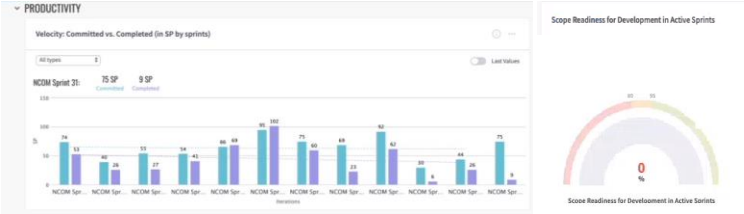
- Enroll Agile related courses for all team members and the client:
  - Agile Thinking
  - Introduction To Scrum
  - Introduction to Kanban
  - Basics of Scaled Agile
- Implement standard framework/methodologies like SCRUM, Kanban or define modified formalized methodology to follow.

Flaws in sprint planning processes

FACTS

- Some tasks in sprint are not estimated
- Team velocity is poorly calculated
- Requirements are not detailed enough before sprint planning meeting
- Sprint planning is not driven by velocity/capacity

METRICS



IMPACT

- Possible missing milestones and deadlines due to wrong/absent estimations
- Low predictability on feature completion due to uncalculated velocity

RECOMMENDATIONS

- Add Tech debt to sprint backlog
- Introduce Data Driven sprint planning process
  - Estimate all tasks in sprint
  - Calculate velocity
  - Use velocity and capacity estimations for sprint planning
  - Analyze Completed vs Committed metrics on retrospective meetings to improve estimations and reach stable velocity
- Prepare user stories in advance
  - Track Scope Readiness (Requirement Readiness) metric
  - Make sure that this metric is more then 100% at each moment of time

EngX

Developers are not involved in technical and business decisions

FACTS

- Only Architect, Dev Lead and BA participate in Three Amigo Sessions to prepare and refine requirements
- Regular team members are not involved in technical decision making
- Tasks are pre-assigned to developers before sprint planning meeting

IMPACT

- There is no team engagement in business requirements and discussions on technical solutions;
- Product quality degradation due to unclear requirements / edge cases;
- Poor team spirit due to low engagement level (team members are Doers only)

RECOMMENDATIONS

- Introduce technical refinement sessions to clarify details of user stories and review acceptance criteria. These sessions should be conducted before sprint planning meetings
- Introduce early technical review sessions to verify technical and architectural decision with project team. It helps to increase engagement level and get early feedback on the decision

There is no Release Plan

FACTS

- Team has no plan for future Release
- No strategy, post-Release actions planned

IMPACT

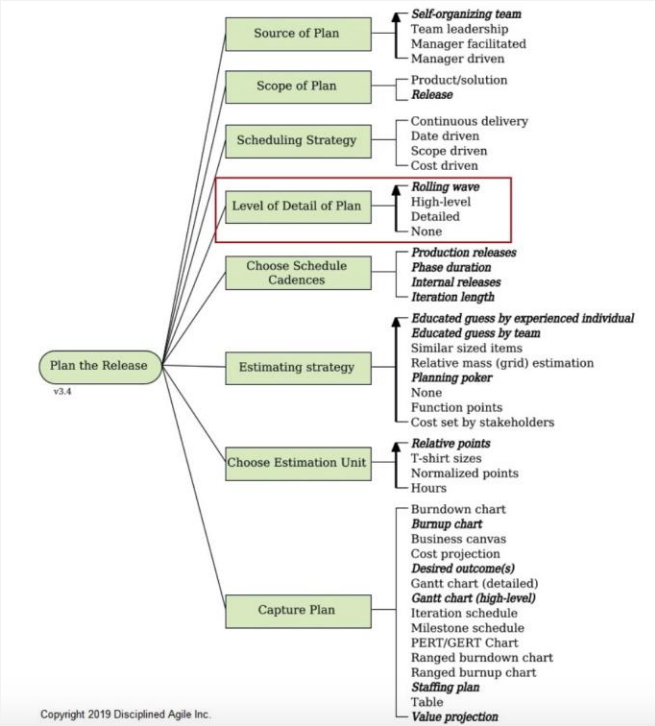
- Without plan will be no ability to think through the scope/functionality as a team and identify dependencies, risk/mitigation and potential impediments
- No goal and measurement of performance benchmarks
- No chance to evaluate priority and discuss on trade-off options

WHY WE NEED?

It is a great mechanism to align all the stakeholders on feature road map, priority, requirements, risk & mitigation plan and schedule. When done right, release planning can ensure there is a solid backlog for the team to work.

RECOMMENDATIONS

See the process goal diagram for how a team may plan the initial release . Notice how a plan may address schedule/time, cost, value, staffing considerations or combinations thereof.



EngX

There is no Release Plan

FACTS

- Team has no plan for future Release
- No strategy, post-Release actions planned

IMPACT

- Without plan will be no ability to think through the scope/functionality as a team and identify dependencies, risk/mitigation and potential impediments
- No goal and measurement of performance benchmarks
- No chance to evaluate priority and discuss on trade-off options

WHY WE NEED?

It is a great mechanism to align all the stakeholders on feature road map, priority, requirements, risk & mitigation plan and schedule. When done right, release planning can ensure there is a solid backlog for the team to work.

RECOMMENDATIONS

Choose the option for planning the release of a solution:

1. **Rolling wave:** The plan is continuously updated throughout the release, lwith more detail for upcoming work, and less detail for work further out.
2. **High-level:** The release plan indicates major milestones, any phases, any iterations/sprints and any dependencies between them. It does not address the detailed work to be performed. Instead it trusts the team to self-organize and do whatever is appropriate at the time.
3. **Detailed:** The release plan contains significant details around the work to be done and may even assign that work to specific roles or people. Details are identified at the beginning of the release, during a period that agile and scrum teams often refer to as “Sprint 0”, Inception or Initiation. The details are typically updated over time as the work proceeds.

Release planning process is not established

FACTS

- Release Scope is not defined in advance;
- There is no release timeline on the project;

IMPACT

- High probability of release failure;
- Decrease agility and flexibility;

PROBLEM STATEMENT

The goal of initial release planning is to estimate roughly which features will be delivered by the release deadline (presuming the deadline is fixed), or to choose a rough delivery date for a given set of features (if scope is fixed);

RECOMMENDATIONS

- **Set up release planning process:**
  - Rank the project backlog;
  - Create project release roadmap;
  - Develop plans for each release;
  - Establish the Definition of Done and Definition of Ready for the release;
- **Articles:**
  - https://kb.epam.com/display/EPMDMO/Release+planning+in+case+of+multiple+vendors+project+delivery
  - https://kb.epam.com/display/EAGILCC/Program+Release+Planning
  - https://kb.epam.com/display/EAGILCC/Release+Plan

Substantial percentage of functionality (up to 35%) get returned into development

FACTS

- Requirements are not clear by the moment of implementation
- Team doesn't pay attention on quantity of returned tasks.
- Team doesn't consider tasks before taking them to sprint and doesn't ask questions.
- Development engineers spend significant effort on QA squad consulting.

IMPACT

- Time & effort waste on switching between planned development and returned tasks
- Unpredictable delivery as a result of working on unexpected tasks
- Team performance could be higher
- Quality is suffering while new team members switch over between planned and sporadic activities
- Growing attrition in a project

PROBLEM STATEMENT

Up to 35% is high number. It mean that teams pending time on getting done tasks that are supposed to be completed. As a result delivery might be unpredictable or unstable. It is impossible to predict how much time the team will spend on working on returned tasks. The team has to switch between them, waste resources on switching and work itself.

RECOMMENDATIONS

- Review definition of ready for development. If there is unclear requirements don't take a task into sprint. Collect open questions and address them to BA/PO.
- Review definition of done. Check whether team follows definition of done and whether definition of done is same for developer and QA.
- Involve QA on refinement/planning sessions so that whole team would be on the same page.

Acceptance Criteria are not detailed

FACTS

- Acceptance criteria in user stories too high level

IMPACT

- Possible rework due to defects related to unclear requirements
- High risk of story rejections due to misunderstanding
- The potential risk of missed critical defects

PROBLEM STATEMENT

Acceptance criteria are the conditions that a software product must meet to be accepted by a user, a customer, or another system. Well-written acceptance criteria helps avoid unexpected results at the end of a development stage and ensures that all stakeholders and users are satisfied with what they get.

Acceptance criteria synchronizes the visions of the client and the development team. They ensure that everyone has a common understanding of the requirements: Developers know exactly what kind of behavior the feature must demonstrate, while stakeholders and the client understand what's expected from the feature.

RECOMMENDATIONS

- Add the presence of Acceptance Criteria to the Definition of Ready
- Agree on the Acceptance Criteria format and use it for each corresponding stories
- Ensure that each developer checks the Acceptance Criteria before completing the task

Too late retrospective meetings

FACTS

- Retrospective meetings happen in the beginning of the next sprint

IMPACT

- Less efficient retrospective because it could be impossible to implement corrective action items in already started sprint

RECOMMENDATIONS

Reschedule Scrum events sequence in the following order:

- Current Sprint Demo/Review
- Current Sprint Retrospective
- Next Sprint Planning

Poor Scrum processes

FACTS

- No communication between PO and team
- Team members could be assigned to the several projects
- No retrospective
- No formalized scope management

IMPACT

- No opportunity for the Scrum team to inspect itself
- No opportunity to clarify requirements with PO

ADDITIONAL INFORMATION

Scrum is good framework by it's constrains and backlog management. Current flow is more likely Kanban process.

RECOMMENDATIONS

- Enroll Agile related courses for all team members and the client:
  - Agile Thinking
  - Introduction To Scrum
  - Introduction to Kanban
  - Basics of Scaled Agile
- Implement standard framework/methodologies like SCRUM, Kanban or define modified formalized methodology to follow.

Lack of communication between teams/vendors

FACTS

- Team is not familiar with processes/knowledge of other teams;
- It takes time to reach right contact points and get feedback;

IMPACT

- Lack of transparency on whole SDLC process which influences project decision;
- Late feedback from other teams;
- No shared responsibilities for the whole project;
- Increase time to market for the product;

PROBLEM STATEMENT

Communication is a source of information to the team members for decision-making process as it helps identifying and assessing alternative course of actions.

Communication also assists in controlling process. It helps controlling organizational member's behavior in various ways.

RECOMMENDATIONS

Get familiar with general SDLC approach between teams:

- what areas of responsibilities are;
- who contact points are;

Set up regular sync up meetings with other teams:

- Resolve dependencies if any
- Check priorities
- Unique approach

Frequent requirements changes

FACTS

- Customer is changing requirements frequently;
- There is no formal change management process on the project

IMPACT

- Frequent development re-work after updating requirements;
- Technical debt is increasing;
- Missing the deadlines;

PROBLEM STATEMENT

If requirements cannot be nailed down at the project, it leads to great inefficiencies later and increases the chances the project will fail.

The earlier the requirements are discovered, then the easier and more efficient it is to plan and design a system that meets those requirements.

RECOMMENDATIONS

- Set up Change Management Process on the project and discuss it with client;
- **Model should be decided a bit ahead:**
  - Using prototypes in advance for discussing requirements in detail before implementation has been started;

  Or/And

  - Use 3 amigo sessions to discuss requirements up front;
- Triage requirements with customer on weekly basis;
- Control scope and steer clear of scope creeping;

Team and Client have not alliance knowledge of Agile practices

FACTS

- There is no formal SDLC approach on the project;

IMPACT

- Unpredictable Costs and Schedule;
- Time to market is increased;
- Losing quality of the product;

RECOMMENDATIONS

- Discuss with the customer possibility to pass Agile training together with the team:
- Gamification:

  https://learn.epam.com/detailsPage?id=19dfcad0-c513-403f-8e4b-dca2153ea1ba
- Courses:

  https://learn.epam.com/detailsPage?id=e02c186c-8c3e-4e40-a31f-c733ad0f2018

The team configuration

FACTS

- The team changes duration of sprints
- Value of Completed vs Committed metric id about 50%

IMPACT

- Low predictability on sprint and release level therefore possible missed customer expectations

RECOMMENDATIONS

- Discuss and document Reference story to ensure that whole team understands what 1 SP is.
- Pass Agile Estimation training
- Conduct mandatory root cause analysis of missed estimation on each retrospective

Scrum ceremonies

FACTS

- Team doesn't estimate backlog items (Story points and/or man hours);
- Team doesn't measure basic metrics (Avg. Velocity, Velocity: Committed vs Completed, Scope creep, Requirements Rewritten)
- Team doesn't have Definition of Done and Definition of Ready.

IMPACT

- While DoD and DoR may be a verbal agreement between team and product owner, there's still a good practice to document it as a project artifacts. It helps to have consistent understanding of Ready for Development and Done independently to change of Scrum Master, Product Owner or team members. It also prevents possible speculation and enforcements to pick up stories are not ready yet just by one side change of rules. Documented DoR may be referenced as a contract what need to be changed and signed first.
- Having measured metrics makes delivery more transparent and predictable

PROBLEM STATEMENT

Unclear criteria for defined and done user stories do not allow to estimate stories.

Team is not able to make data driven decisions, for instance, plan a sprint scope.

Lack of basic metrics makes obscure project delivery results, less transparency and predictability.

Lack of measured metrics makes release planning vague.

RECOMMENDATIONS

- Introduce Definitions of Done and Ready for Development:
  - DoD example: https://kb.epam.com/download/attachments/343682868/Definition_of_Done.doc?version=1&modificationDate=1470671392000&api=v2
  - **DoR best practices.** DoR example in Appendix A.
- Introduce **effort pointing** practice in sprint planning ceremony;
- Introduce **Performance Monitor** to measure key metrics of team's productivity, product quality and project progress:
  - Avg. Velocity;
  - Velocity: Committed vs Completed;
  - Scope creep;
  - Requirements Rewritten;
  - Bugs: Open vs Resolved;
  - Project BurnUp.

Waterfall development model is used

FACTS

- The Customer follows Waterfall development model
- Requirements can be changed multiple times during a day.
- The team tries to apply agile practices in their everyday work;

IMPACT

- Time & effort waste due to lack of agility/flexibility in requirements changes
- Outdated development approach reduces employee satisfaction rate
- Lack of capabilities to improve current processes/performance due to long iterations/releases

RECOMMENDATIONS

- Consider running Scrum inside Waterfall. Apply Scrum practices inside EPAM team
- Involve customer in Agile value trainings. Discuss Agile transformation of the project.

The team is unstable. High attrition rate.

FACTS

- **[Aliens]** Team was recomposed totally 100% for last 6 months (left EPAM, relocation, part of technologies are outdated).
- **[Predators]** Unstable team (stability evaluated as 2 of 5)

IMPACT

- Time & effort waste on new team members onboarding
- Lost knowledge when experienced team members leave
- Quality is suffering while new team members learn the code base & domain
- Velocity is reduced while new team member is getting up to speed

PROBLEM STATEMENT

High attrition is a huge issue impacting all aspects of project – team motivation, team performance, client's satisfaction. It's crucial to work on preventing / reducing attrition and on mitigating consequences where attrition is unavoidable.

Keeping high team morale and creating attractive working environment are major keys to preventing people from leaving.

RECOMMENDATIONS

- Reduce attrition. Some basic recommendations:
  - Look for technically challenging (research, optimization) tasks
  - Reserve part of sprint/release capacity for research tasks
  - Work on team building & team spirit, secure comfortable and interesting working environment for team members
- Mitigate attrition
  - Ensure documentation is maintained and always up to date
  - Use succession planning to secure knowledge
- Read more hints here: https://kb.epam.com/display/EPMDMO/How+to+decrease+attrition+at+business+unit+level

Sprints can be started without refined backlog

FACTS

- Sometimes backlog refinement session can be conducted after sprint start

IMPACT

- Story development and testing may take more time than it was planned and affect sprint deliverables and commitments.
- Rework may be needed in case of any changes in requirements
- Potential risks to affect product/code quality because of changes and possible time pressure in development and testing

PROBLEM STATEMENT

Product Backlog Refinement is not for stories selected for the current Sprint; it is for items in future Sprints. A good practice is to have at least two Sprints worth of work ready to go in the Product Backlog. Sprint Planning becomes relatively simple because the Product Owner and Scrum Team start the planning with a clear, well analyzed and carefully Estimated set of stories. If refining the Backlog is not being done (or not being done well) Sprint Planning will involve a significant amount of questions, discovery and or confusion.

RECOMMENDATIONS

- Plan refinement sessions in advance before the sprint starts
  - Define constant cadence for all sprint events
- Make a retrospective for each case when refinement wasn't finished till new sprint start
- (Optional) Track time of rework because of unclear and not clarified in advanced requirements

Definition of Done and Ready is not used

FACTS

- There are written description of DoD and DoR, but it is not used during story acceptance.
- Team members are not informed about DoD and DoR

IMPACT

- Product quality could be degraded because some important quality checks are missed
- Possible rework due to late defects identification
- Possible rework or idle time due to story incompleteness for development

PROBLEM STATEMENT

The Definition of Done ensures everyone on the Team knows exactly what is expected of everything the Team delivers. It ensures transparency and quality fit for the purpose of the product and organization. Some researchers say that getting stories done can double a Teams Velocity.

Getting started with a poorly understood story can create many roadblocks for scrum team. A story without proper information can lead to rework of work at best or work which takes the completely wrong direction at worst. It's so clear that a user story has to meet a set of minimum criteria before it's ready for inclusion in the work of the next sprint. This set of minimum criteria is the Definition of Ready and, like the Definition of Done, should be agreed upon by the Scrum team. This shared definition then allows the team to reject the stories that don't have clearly defined acceptance criteria. It will save a lot of time if each user story meets Definition of Ready before the Sprint Planning meeting.

RECOMMENDATIONS

- Discuss understanding of DoD and DoR with Product Owner, Scrum Master, Team.
- Create DoD and DoR document it in KB.
- Ask stakeholders to approve/sign-off it
- Use DoR during Sprint Planning
- Use DoD during story implementation, bug fixing, sprints, releases, etc.

Acceptance Criteria are not mandatory for user stories

FACTS

- Some stories do not have acceptance criteria
- Acceptance criteria format is not defined

IMPACT

- Possible rework due to defects related to unclear requirements
- High risk of story rejections due to misunderstanding
- The potential risk of missed critical defects

PROBLEM STATEMENT

Acceptance criteria are the conditions that a software product must meet to be accepted by a user, a customer, or another system. Well-written acceptance criteria help avoid unexpected results at the end of a development stage and ensure that all stakeholders and users are satisfied with what they get.

Acceptance criteria synchronize the visions of the client and the development team. They ensure that everyone has a common understanding of the requirements: Developers know exactly what kind of behavior the feature must demonstrate, while stakeholders and the client understand what's expected from the feature.

RECOMMENDATIONS

- Add the presence of Acceptance Criteria to Definition of Ready
- Agree on Acceptance Criteria format and use it for each corresponding stories
- Ensure that each developer checks Acceptance Criteria before task completion

Teams do not have visibility of release progress

FACTS

- Teams don't know the release management process and when the story will be released to production
- Release burndown/burnup charts are not used

IMPACT

- Potential changes in sprint backlog because of the fix of urgent issues
- Low possibility to propose process improvements without an understanding of overall release process
- Late quality feedback not allowed to improve the development process and testing strategy

PROBLEM STATEMENT (ORTIONAL)

Visibility of a project drives team performance, task ownership, accountability, and better communication.

When the entire project—and everyone's task and responsibility—is visible to the entire team, people rise to a level of accountability and performance that doesn't happen when lost in invisible silos.

Transparency also holds individuals accountable and motivates them to deliver on their commitments.

RECOMMENDATIONS

- Clarify the release management process
- Plan sprint backlog in accordance with current release state

Sprint velocity is not constant

FACTS

- Development engineers do not have sufficient visibility of QE processes.
- Team doesn't measure velocity.
- Team members switch over between different work streams.
- Development engineers spend significant effort on QA squad consulting.

IMPACT

- Time & effort waste on switching between planned development and sporadic QA consultancy activities
- Declining team motivation
- Unattractive project environment for newcomers
- Quality is suffering while new team members switch over between planned and sporadic activities
- Growing attrition in a project

PROBLEM STATEMENT

Team combines 2 conflicting engagement models, planned and sporadic activities. It influences all aspects of project – team motivation, team performance, quality of deliverables. It's crucial to work on preventing / reducing number of conflicting types of activities and on mitigating consequences where conflicts are inevitable.

Keeping high team morale and creating attractive working environment are major keys to keep high morale in the team and prevent people from leaving.

RECOMMENDATIONS

- Introduce Kanban practice and allocate engineering resources for processing issues that were not planned in scope of Sprint (consulting QA engineers, reproducing bugs, fixing immediate issues);
- Introduce EPAM Quality Engineer in QA squad to ensure connectivity between EPAM development and customer's QA teams;
- Reduce attrition. Some basic recommendations:
  - Look for technically challenging (research, optimization) tasks
  - Reserve part of sprint/release capacity for research tasks
  - Work on team building & team spirit, secure comfortable and interesting working environment for team members
- Mitigate attrition
  - Ensure documentation is maintained and always up to date
  - Use succession planning to secure knowledge
- Read more hints here: https://kb.epam.com/display/EPMDMO/How+to+decrease+attrition+at+business+unit+level

Engineering

Low technical debt management process

FACTS

- Unintentional technical debt is prevented by linters
- Intentional technical debt is not tracked
- Intentional technical debt is not estimated

IMPACT

- Architecture becomes more loosely-coupled
- Possible code quality degradation
- Lower maintainability

RECOMMENDATIONS

- Set up a process of tracking technical debt on a project:
  - Track intentional technical debt on JIRA as tasks
  - Estimate these tasks
  - Take some of them into sprints in order to reduce level of intentional technical debt
- Describe each process on Confluence or KB space

Inconsistent engineering practices across different repositories

FACTS

- Teams manage about 50 repositories
- There is no an automated way to setup CI/CD infrastructure for new repositories
- CI/CD pipelines and used tools are different for different repositories

ARTIFACTS

Repository type	Repository name	Owner	CI	CD	Auto testing	Static code analysis	Unit test coverage
Main CRM repo	**CRM**	all	+	+	+/-	-	47,6%
CRM UI test automation repo	**CRM automation**	all qe	-	N/A	N/A	-	N/A
New APIs	**integration-aws-reporttransmission**	Stargazer	+	+	sa e-2-e	-	?
	fila-provider-service	Stargazer	+	+	sa	-	?
	fila-provider-service-authorizer	Stargazer	+	+	sa	-	? ~50%
	D365.AzureFunctions	Stingray	+	-	sa e-2-e	+	? ~50%
Roche integration	**roche-integration-event-sender**	(Qualia) other tear	+	+	+	+	98,4%
	roche-integration-receiver	(Qualia)	+	+	+	+	100,0%
	roche-integration-sender	(Qualia)	+	+	+	+	98,4%
	roche-integration-case-service	(Qualia)	+	+	+	+	92,2%
	roche-integration-mock	(Qualia)	+	+	N/A	-	? ~50%
	roche-integration-cms	(Qualia)	+	+	+	+	65,7%
	roche-integration-testing-common	(Qualia)	+	N/A	N/A	-	? 20%
	roche-integration-service	(Qualia)	+	+	+	+	98,5%
	roche-integration-common	(Qualia)	+	N/A	+	+	85,9%
	roche-integration-response-handler	(Qualia)	+	+	+	+	96,9%
	roche-integration-order-publisher	(Qualia)	+	+	+	+	90,3%
	roche-integration-case-publisher	(Qualia)	+	+	+	+	96,1%
	roche-integration-order-service	(Qualia)	+	+	+	+	93,8%
	roche-integration-support-request-publisher	(Qualia)	+	+	+	+	99,0%
	roche-integration-e2e-tests	(Qualia)	+	N/A	N/A	-	N/A
Old APIs	**integration-ui**	all	-	-	-	-	0
	transaction-api	all	-	-	-	-	0
	query-api	all	-	-	-	-	0
	crm.integration.query.api	all	-	-	-	-	? maybe 10%
	inbound-service	all	-	-	-	-	0
	outbound-service	all	-	-	-	-	0
	crm.integration.data	all	-	-	-	-	? maybe 10%
Tools (more than 20 repos)	**Data contracts**	no team	N/A	N/A	N/A	-	N/A
	SSIS	no team	-	-	-	-	N/A
	one time tools	no team	N/A	N/A	?	-	N/A
Others	**<different names>**	no team	-	-	-	-	?

IMPACT

- Code and product quality degradation
- Less efficient development and testing activities

RECOMMENDATIONS

- Increase System Engineers' capacity. Consider extending a team by adding a System/Build engineer for implementation of recommendations below:
  - Design **TO BE** state of CI/CD infrastructure for each type of repository (CRM, Service, Automation, etc.)
  - Consider implementation of an automated approach to provision desired CI/CD infrastructure for newly created repositories.
  - Unify CI/CD infrastructure for each repository (manually or automatically)

Commit message template is not defined

FACTS

- There is no unified commit message template on the project

IMPACT

- Work on commit history is more difficult and less systematic (i.e. readability, search-ability, debugging, investigating issues, etc.)
- Higher the entry level for new contributors (difficult to follow)

PROBLEM STATEMENT

A project's long-term success rests (among other things) on its maintainability, and a maintainer has few tools more powerful than his project's history revision.

In order to increase readability and maintainability of created code – commit message template is a good option, that should allow to see all commits associated with specific user story/feature or defect.

RECOMMENDATIONS

- Define uniquee commit message template and discuss it with all stakeholders;
- Follow the best practices of great Git commit message:
  - Separate subject from body with a blank line;
  - Limit the subject line to 50 characters;
  - Capitalize the subject line;
  - Do not end the subject line with a period;
  - Use the imperative mood in the subject line;
  - Wrap the body at 72 characters;
  - Use the body to explain ***what*** **and** ***why*** **vs.** ***how***
- Describe defined process on the Confluence or KB space;

There is no architecture significant requirements

FACTS

- There is no list of Architecture Significant Requirements in Solution Architecture documentation

IMPACT

- Lack of Architecture significant requirements could significantly increase costs of changes (The cost can be: money, time, resources, reputation, etc.)

ADDITIONAL INFORMATION

An architecturally significant requirement (ASR) is a requirement that will have a profound effect on the architecture—that is, the architecture might well be dramatically different in the absence of such a requirement.

RECOMMENDATIONS

- Gather ARS and verify all architecture/design related decisions against ARS
- EPAM Software architecture School could be a good set of trainings to improve your architecture

There are no CI jobs on feature branches and Pull Requests

FACTS

- The earliest CI job is executed on Master branch

IMPACT

- Possible rework in the Code Review process due to the review of the code that will not pass technical quality gates
- Late feedback from CI

PROBLEM STATEMENT

Feature Branch/Pull Request verification job prevents intervention of bad code onto common branch. It is important to keep common branch in “green” state. A defect in common branch can lead to its propagation to derived branches, or even block further development and testing activities.

RECOMMENDATIONS

- Introduce CI job which is triggered by Pull Request creation/update or push in Feature Branch
- Include in the job as much verifications as possible but limit its duration to 30 minutes (< 10 minutes preferred). Recommended verifications:
  - Unit Tests
  - Code Coverage
  - Static Code Analysis (Sonar)
  - Smoke Tests
- As a first step, use results of the job as a manual quality gate for Pull Request acceptance
- Once PR quality is a stable set technical limitation to accept a pull request in case of a failed job.

There is no newcomer's adaptation process/guide

FACTS

- There is no handbook/guide to learn about product, team structure and development process set up;

IMPACT

- Communication waste inside the team;
- Rework after wrong feature implementation by newcomer;
- Low velocity of task implementation by newcomer
- Long learning process;

PROBLEM STATEMENT

Project onboarding is a complex process which requires a lot of efforts both from a newcomer and a team. Frequently newcomer cannot catch general design principles and rules on the project. It is hard to infer high level design principles from given realization only without proper documentation and expert knowledge.

RECOMMENDATIONS

- Prepare an adaptation plan:
  - Describe a general information about the project;
  - Define a stakeholders' list;
  - Describe development process;
  - Tools and technics are used in the team should be mentioned;
  - Include coding standards and code review process in newcomers guide;
- Assign mentor for each newcomer;
- Newcomers should address all questions to their mentors only;
- If mentor cannot answer a question, he/she can forward it to right team member.
- Mentor should provide feedback on the following areas:
  - To newcomer regarding his overall performance
  - To team regarding possible improvements of onboarding process
- Document this process in Confluence or on kb

There is no structured Logging and Monitoring approaches followed

FACTS

- No logging libraries are used;
- There are no tools for collection and analyzing logs;
- No monitoring approach is followed on the project;

IMPACT

- Root Cause Analysis for production issues is time consuming due to lack of tools for processing logs;
- Late issues' identification because of missing monitoring approach on the project;

RECOMMENDATIONS

- Select libraries for logging (like Nlog or log4Net)
- Document logging approach (how to use selected libraries)
- Set up special tools like ELK stack to investigate issues and analyze logs
- Consider implementation of support interfaces or API to show status indicators for load or performance;
- Set up special tools like Zabbix for infrastructure and application monitoring

Prudent technical debt is not tracked on the project

FACTS

- Prudent technical debt is not tracked
- Prudent technical debt is not estimated

IMPACT

- Tech is diversifying;
- Architecture is becoming more loosely-coupled;
- Possible code quality degradation;

PROBLEM STATEMENT

Technical debt is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.

Prudent Deliberate debt represents quality shortcuts made consciously by the team because of an imminent deadline. The team recognizes the issue and its consequences but must choose to deliver something rather than focus on quality at this time.

RECOMMENDATIONS

- Train engineers if they have not enough knowledge https://learn.epam.com/detailsPage?id=66f15bb9-4358-4e1a-aad9-370406c2b0ce
- Set up a process of tracking technical debt on a project:
  - Include measurement of tech debt into CI;
  - Analyze results on a daily/weekly basis;
  - Track prudent technical debt in JIRA;
  - Estimate prude technical debt;
  - Take some of them into sprints in order to reduce level of prudent technical debt;
- Describe a process on Confluence or KB space;

Commit message template is not defined (nice to have)

FACTS

- There is no unified commit message template on the project

IMPACT

- Work on commit history is more difficult and less systematic(i.e. readability, search-ability, debugging, investigating issues, etc.)
- Higher the entry level for new contributors (difficult to follow)

PROBLEM STATEMENT

A project's long-term success rests (among other things) on its maintainability, and a maintainer has few tools more powerful than his project's history revision.

In order to increase readability and maintainability of created code – commit message template is a good option, that should allow to see all commits associated with specific user story/feature or defect.

RECOMMENDATIONS

- Define uniquee commit message template and discuss it with all stakeholders;
- Follow the best practices of great Git commit message:
  - Separate subject from body with a blank line;
  - Limit the subject line to 50 characters;
  - Capitalize the subject line;
  - Do not end the subject line with a period;
  - Use the imperative mood in the subject line;
  - Wrap the body at 72 characters;
  - Use the body to explain what and why vs. how
- Describe defined process on the Confluence or KB space;

No Commit Message template (minor)

FACTS

- There is no formal or informal commit message template
- Some commits do not have reference to corresponding JIRA issue

IMPACT

- Lost of traceability between source code and reason of the change
- Increased complexity of code maintenance

PROBLEM STATEMENT

Commits history is a tool to identify true reasons and context why the particular code was written. So it is important to clarify developer's intentions in commit messages. It helps to simplify code review process because it becomes much more clear why the code change was done and verify the author decisions. Also sometimes efforts on new feature development and bug fixing could be reduced by adding additional context to the code via commit messages.

The first step to reveal the reason behind he change is to link each commit with corresponding tasks in task tracking system. In this case it will be more clear why the code change is happened.

RECOMMENDATIONS

- Introduce commit message guidelines, rules and templates.
  - Ensure that traceability to corresponding task is included in the template
- Verify commit messages during code review process. (Add this point to code review checklist, if any)

Example of commit message rules:

- Separate subject from body with a blank line
- Subject should contain key/id and summary of JIRA issue.
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters
- Use the body to explain what and why vs. how

Example of Commit message guideline

Using different VCSes (SVN, Git) in the project

FACTS

- New source code is stored in Git repositories;
- General VCS is SVN for teams;
- There is lack of practical skill of Git usage in team;

IMPACT

- Difficulties to keep Source Code in several VCSes up to date;
- Lack of integrity between old and new feature development;
- Waste team time on switching between VCS types;
- Delay with delivery;
- Code duplication;
- SVN has no longer a vibrant community supporting;

RECOMMENDATIONS

- Set up migration from SVN to Git:
  - Prepare a migration environment;
  - Convert the source SVN repository to a local Git repository
  - (Optional) Synchronize the local Git repository with any changes from SVN repository while developers continue using SVN
  - Push the local Git repository to a remote Git repository
  - Switch Developers to Git as main source control system
  - https://www.atlassian.com/git/tutorials/migrating-overview
- Document this process in Confluence or on kb

There is lack of guards in SDLC

FACTS

- Not all of the changes walk through the default delivery flow. Some changes may bypass Feature branch and Pull Request stages;
- Team member is ably merge changes without Pull Request approval;
- Regardless of the results of BlackDuck and SCA checks on Dev branch change may be deployed to integration environment.

PROBLEM STATEMENT

Quality gates prevent defects propagations. Not passed quality gate means that change's source code does not fulfill quality agreement and must not be used further without rework. Usually it's implemented as red CI pipeline, which must be fixed immediately to continue development. CI tools usually enforce quality gates via automation, but it can be checked manually too.

RECOMMENDATIONS

- Restrict merging changes from local development environment directly to dev branch bypassing "Feature branch" and "Pull Request" stages;
- Fail build-deploy process if quality checks are failed, for example, restrict deployment to Integration environment if BlackDuck and SCA checks were failed;
- Introduce "rule of thumb" for each Pull Request to facilitate peer code review practice in the team;
- Configure Git to restrict merges that were not approved by the other team members.

Absence of Pull Request template

FACTS

- There is no formal Pull Request message template /Commit message template
- There is informal rule that each commit should contain link to JIRA issue

IMPACT

- Possible lost of traceability
- Increased complexity of code maintenance

PROBLEM STATEMENT

Commits history is a tool to identify true reasons and context why the particular code was written. So it is important to clarify developer's intentions in commit messages. It helps to simplify code review process because it becomes much more clear why the code change was done and verify the author decisions. Also sometimes efforts on new feature development and bug fixing could be reduced by adding additional context to the code via commit messages.

RECOMMENDATIONS

- Introduce commit message guidelines, rules and templates.
- Verify commit messages during code review process. (Add this point to code review checklist)

Example of commit message rules:

- Separate subject from body with a blank line
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters
- Use the body to explain what and why vs. how

Example of Commit message guideline

Absence of Quality gates on Pull Request

FACTS

- There is no Code Coverage and Static Code Analysis quality gates which prevents to merge Pull Request in case of poor code and tests quality

IMPACT

- Possible rework in the Code Review process due to the review of the code that will not pass technical quality gates
- Low quality of the code in Develop branch due to absence of pre commit quality gates leads to unstable Development environment
- Merge of new features into broken common branch hides defects in merged functionality

PROBLEM STATEMENT

Pull Request verification job prevents intervention of bad code onto common branch. It is important to keep common branch in “green” state. A defect in common branch can lead to its propagation to derived branches, or even block further development and testing activities.

RECOMMENDATIONS

- Introduce CI job which is triggered by Pull Request creation/update
- Include in the job as much verifications as possible, but limit its duration to 30 minutes. Recommended verifications:
  - Unit Tests
  - Code Coverage (See other recommendations)
  - Static Code Analysis (See other recommendations)
  - Automated Smoke Tests (See other recommendations)
- As a first step, use results of the job as a manual quality gate for Pull Request acceptance
- Once Pull Request quality is stable, set technical limitation to accept a pull request in case of a failed job.

No Commit Message template

FACTS

- There is no formal or informal commit message template
- Some commits do not have reference to corresponding JIRA issue

IMPACT

- Lost of traceability between source code and reason of the change
- Increased complexity of code maintenance

PROBLEM STATEMENT

Commits history is a tool to identify true reasons and context why the particular code was written. So it is important to clarify developer's intentions in commit messages. It helps to simplify code review process because it becomes much more clear why the code change was done and verify the author decisions. Also sometimes efforts on new feature development and bug fixing could be reduced by adding additional context to the code via commit messages.

The first step to reveal the reason behind he change is to link each commit with corresponding tasks in task tracking system. In this case it will be more clear why the code change is happened.

RECOMMENDATIONS

- Introduce commit message guidelines, rules and templates.
  - Ensure that traceability to corresponding task is included in the template
- Verify commit messages during code review process. (Add this point to code review checklist, if any)

Example of commit message rules:

- Separate subject from body with a blank line
- Subject should contain key/id and summary of JIRA issue.
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters
- Use the body to explain what and why vs. how

Example of Commit message guideline

Auto tests are not executed on Pull Request

FACTS

- The earliest CI job is executed on Master branch
- Customer already rejected request to create new environment

IMPACT

- Possible rework in the Code Review process due to the review of the code that will not pass technical quality gates
- Work stoppage (idle time) because it is impossible to merge branches while the defect in the common branch is not fixed

PROBLEM STATEMENT

Pull Request verification job prevents intervention of bad code onto common branch. It is important to keep common branch in “green” state. A defect in common branch can lead to its propagation to derived branches, or even block further development and testing activities.

RECOMMENDATIONS

- Add execution of API/E2E test to Pull Request verification job.
- If necessary, re-discuss with customer preparation of separate environment to run Pull Requests on it.

There is no possibility to test feature on AWS environment from feature branch

FACTS

- It is impossible to deploy to “real” AWS environment from feature branches
- Some features could have dependency on AWS infrastructure, so it is impossible to properly test it on local environment
- Customer already rejected request to create new environment

IMPACT

- Work inefficiency due to unnecessary steps (like merge request creation, code review, etc.) to perform development testing on AWS environment

RECOMMENDATIONS

- Introduce CI job which can be run on any feature branch to deploy code from the branch to AWS environment
- If necessary, re-discuss with customer introduction of one more AWS environment for feature branches only to prevent possible interference of not finalized feature from feature branch to stable development environment.

Absence of Pull Request template

FACTS

- There is no formal Pull Request message template /Commit message template
- There is informal rule that each commit should contain link to JIRA issue
- Team squashes commits from feature branches before merge to master branch

IMPACT

- Possible lost of traceability
- Increased complexity of code maintenance

PROBLEM STATEMENT

Commits history is a tool to identify true reasons and context why the particular code was written. So it is important to clarify developer's intentions in commit messages. It helps to simplify code review process because it becomes much more clear why the code change was done and verify the author decisions. Also sometimes efforts on new feature development and bug fixing could be reduced by adding additional context to the code via commit messages.

RECOMMENDATIONS

- Introduce commit message guidelines, rules and templates.
- Verify commit messages during code review process. (Add this point to code review checklist)

Example of commit message rules:

- Separate subject from body with a blank line
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters
- Use the body to explain what and why vs. how

Example of Commit message guideline

There is no Pull Request verification job

FACTS

- The earliest CI job is executed on Master branch

IMPACT

- Possible rework in the Code Review process due to the review of the code that will not pass technical quality gates
- It will be impossible to create new branch while the defect in the common branch is not fixed
- Merge of new features into broken common branch hides defects in merged functionality

PROBLEM STATEMENT

Pull Request verification job prevents intervention of bad code onto common branch. It is important to keep common branch in “green” state. A defect in common branch can lead to its propagation to derived branches, or even block further development and testing activities.

RECOMMENDATIONS

- Introduce CI job which is triggered by Pull Request creation/update
- Include in the job as much verifications as possible, but limit its duration to 30 minutes. Recommended verifications:
  - Unit Tests
  - Code Coverage (See next slide)
  - Component Tests
  - Integration Tests
  - E2E Smoke test
- As a first step, use results of the job as a manual quality gate for Pull Request acceptance
- Once PR quality is a stable set technical limitation to accept a pull request in case of a failed job.

No Commit Message template (minor)

FACTS

- There is no formal or informal commit message template
- Some commits do not have reference to corresponding JIRA issue

IMPACT

- Lost of traceability between source code and reason of the change
- Increased complexity of code maintenance

PROBLEM STATEMENT

Commits history is a tool to identify true reasons and context why the particular code was written. So it is important to clarify developer's intentions in commit messages. It helps to simplify code review process because it becomes much more clear why the code change was done and verify the author decisions. Also sometimes efforts on new feature development and bug fixing could be reduced by adding additional context to the code via commit messages.

The first step to reveal the reason behind he change is to link each commit with corresponding tasks in task tracking system. In this case it will be more clear why the code change is happened.

RECOMMENDATIONS

- Introduce commit message guidelines, rules and templates.
  - Ensure that traceability to corresponding task is included in the template
- Verify commit messages during code review process. (Add this point to code review checklist, if any)

Example of commit message rules:

- Separate subject from body with a blank line
- Subject should contain key/id and summary of JIRA issue.
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters
- Use the body to explain what and why vs. how

Example of Commit message guideline

Unit tests creation process is missed

FACTS

- Unit test creation process was lost ;
- Unit Test coverage is decreasing day by day;

IMPACT

- Code quality and Maintainability degradation;
- There is no way to refactor the code;
- Missed defects;
- The less testable code in the future;

PROBLEM STATEMENT

Unit tests coverage is an important metric to understand how well code is covered with unit tests. This information is especially useful when complex conditional logic with many alternative flows is tested. Code coverage is measured not only to understand current status but to measure and control team progress. Ideal 100% coverage is not a goal because the exact threshold depends on many parameters like a technological stack, programming language, % of legacy code, etc.

Reviewing the code coverage result helps to identify code path(s) that are not covered by the tests. This information is important to improve the test collateral over time by reducing the testing debt.

RECOMMENDATIONS

- Resume Unit tests creation process for new code development;
- Add creation of Unit test in User Stories/Features DoD;
- Train engineers if they have not enough knowledge https://learn.epam.com/detailsPage?id=1814e28c-4c17-4f36-b053-e874608d818b
- Agree with the team about code coverage for new code;
- Add code coverage on CI;
- Setup code coverage quality gate on CI
  - Ideally, it should be a quality gate on NEW code only;
- Use code coverage information during code review to identify gaps in unit test cases;

Lack of Unit tests

FACTS

- There is no unit test for Front End part;
- Code Coverage for back end is 10%;
- Code Coverage target is not set

IMPACT

- Code quality and Maintainability degradation;
- There is no way to refactor the code;
- Missed defects;
- The less testable code in the future;

PROBLEM STATEMENT

Unit tests coverage is an important metric to understand how well code is covered with unit tests. This information is especially useful when complex conditional logic with many alternative flows is tested. Code coverage is measured not only to understand current status but to measure and control team progress. Ideal 100% coverage is not a goal because the exact threshold depends on many parameters like a technological stack, programming language, % of legacy code, etc.

Reviewing the code coverage result helps to identify code path(s) that are not covered by the tests. This information is important to improve the test collateral over time by reducing the testing debt.

RECOMMENDATIONS

- Resume Unit tests creation process for new code development;
- Add creation of Unit test in User Stories/Features DoD;
- Train engineers if they have not enough knowledge. Working Effectively with Legacy Code could be a good start
- Agree with the team about code coverage for new code;
- Add code coverage on CI;
- Setup code coverage quality gate on CI
  - Ideally, it should be a quality gate on NEW code only;
  - As the first step, Code Coverage quality gate could be applied manually (no PR approval)
- Use code coverage information during code review to identify gaps in unit test cases;

Lack of Unit testing on front-end

FACTS

- There is no unit testing for front-end part (11 tests only);
- There are 3000+ unit tests on back end;

IMPACT

- Code quality and Maintainability degradation;
- There is no way to refactor the code safely;
- Missed defects;
- The less testable code in the future;

PROBLEM STATEMENT

Unit tests coverage is an important metric to understand how well code is covered with unit tests. Code coverage is measured not only to understand current status but to measure and control team progress. Ideal 100% coverage is not a goal because the exact threshold depends on many parameters like a technological stack, programming language, % of legacy code, etc.

Reviewing the code coverage result helps to identify code path(s) that are not covered by the tests. This information is important to improve the test collateral over time by reducing the testing debt.

RECOMMENDATIONS

- Resume/introduce Unit tests creation process for **NEW** front-end code development;
- Add creation of Unit test in User Stories Definition of Done;
- Train engineers if they do not have enough knowledge. Working Effectively with Legacy Code could be a good start
- Agree with the team about code coverage for **NEW** code;
- Add code coverage calculation in CI;
- Setup code coverage quality gate on CI
  - Ideally, it should be a quality gate on NEW code only;
  - As the first step, Code Coverage quality gate could be applied manually (no PR approval)
- Use code coverage information during code review to identify gaps in unit test cases.

There is lack of unit tests coverage

FACTS

- GlobalAtlas is a platform with a huge legacy code base which is hard to cover with unit tests;
- Team doesn't include effort on unit tests development when estimates user stories.
- Current unit test coverage is 0% (calculated by SonarQube for group of repositories that are related to Global Atlas)

IMPACT

- Unit tests provide fastest feedback among other test activities in dev-test cycle. Having unit tests in place reduces feedback loop for a developer.
- Having unit tests in place reduces developer's effort on extension existing functionality w/o risk of breaking existing functions covered with unit tests.

PROBLEM STATEMENT

Unit tests are important part of test automation pyramid. Their purpose is to cover every idea implemented by developer with set of tests and prove that implementation is correct both from business and technical perspective. Unit tests improve maintainability of the source code protecting it from mistakes and providing easy way to see how/why code works in particular way. In addition developers get great tool for fast feedback from any code changes. Having good coverage of unit tests number of higher level tests may be significantly reduced simplifying testing approach and saving time on testing everything at all levels.

RECOMMENDATIONS

- Sell unit testing ideas to the team, make sure everybody understands the value of this practice for their daily job:
  - https://library.epam.com/minsk/books/search/Working%20Effectively%20with%20Legacy%20Code/28569
- Choose and discuss appropriate unit testing toolset with the team, so tests could be easily created and supported at any level
- Add unit tests as mandatory artifact in Definition of Done (DoD)
- Extend code review checklist to verify unit tests presence and enough coverage is there for every task
- Add unit tests as quality gateway to CI/CD pipeline for new features, make sure developers run them as a part of local check that task is done and ready for integration:
  - https://docs.sonarqube.org/7.4/user-guide/fixing-the-water-leak/
- Cover following topics on regular educational events or centralized trainings: basics of test design, TDD, testing of different layers (DB, MVC, UI, etc.), mocks usage...
- Setup "inspect and adapt" continuous improvement cycle to make this practice really useful for everybody in the team.

Unit test code coverage is not measured in IDE

FACTS

- Unit Test code coverage 90+%
- Code Coverage is calculated on CI only

IMPACT

- It becomes harder to check quality of just written unit tests if it is impossible to immediately check which line of code is not covered by unit tests

RECOMMENDATIONS

- Enable IntelJ IDEA code coverage functionality for every developers' IDE
- Introduce pre-commit "manual" rule to check which lines of just written code is covered by unit tests and improve unit tests suite if some non-trivial code is not covered.

Unit test code coverage is not measured

FACTS

- The team writes unit tests when it is applicable
- Code coverage is not calculated and target is not set

IMPACT

- There is a possibility that the unit test suite is not enough to cover all relevant test cases which lead to:
  - Missed defects
  - More complex refactoring
  - The less testable code in future

PROBLEM STATEMENT

Unit tests coverage is an important metric to understand how well code is covered with unit tests. This information is especially useful when complex conditional logic with many alternative flows is tested. Code coverage is measured not only to understand current status but to measure and control team progress. Ideal 100% coverage is not a goal because the exact threshold depends on many parameters like a technological stack, programming language, % of legacy code, etc.

Reviewing the code coverage result helps to identify code path(s) that are not covered by the tests. This information is important to improve the test collateral over time by reducing the testing debt.

RECOMMENDATIONS

- Taking in account data-centric project nature agree with the team what code should be covered by unit tests.
- Structure your code base for easy automatic identification of “to be covered” code
- Agree with the team about minimum code coverage level for “to be covered code” (60%+ recommended)
- Setup code coverage quality gate on CI
- Measure code coverage on CI Merge Request Verification job (See the previous slides)
- Use code coverage information during code review to identify gaps in unit test cases.

Unit tests quality is not clear

FACTS

- Current unit tests coverage is up to 80%
- Quality of unit tests is important for the team
- Unit tests are creating having completed feature code that violence FIRST principles

IMPACT

- Unclear (opaque) unit tests quality and therefore possible lowered product quality

PROBLEM STATEMENT

Code coverage is used to understand problems in unit tests. Code coverage doesn't show how good the tests are; it shows how bad are they. For example if test coverage is 50%, it means that around 50% of functionality is not tested properly and even if all happy paths are covered, there're still a lot of alternative and negative scenarios, which could not work as intended. At the same time, if code coverage is 90%, it doesn't mean we are certainly safe, high coverage could be achieved by easy to write tests, which covers really nothing. High code coverage target may push developers to satisfy target instead of carefully test functionality.

RECOMMENDATIONS

- To check unit tests' quality – may introduce Mutation testing:
  - Define if Mutation testing is applicable for the project
  - Choose tool for mutation testing (recommendation: https://visualmutator.github.io/web)
  - Execute mutation tests for repositories with code coverage more that 80%
  - Analyze results and define if it is necessary improve unit testing approach for your teams
  - Create technical debt tasks to fix gaps in unit tests for most critical areas
  - Apply steps above on regular basis (per sprint ideally)
  - Consider incorporation of mutation tests in Delivery (CI\CD) Pipeline

Measurement of unit test coverage is not included in Continuous Integration process

FACTS

- Unit test coverage is not measuring in CI pipeline
- Continuous Integration build is not broken in case of unit test code coverage is not met

IMPACT

- Unit tests can be skipped in rush of development;
- Level of code quality degrades;
- If tests are missed, broken code is in repository in feature;
- There is different code coverage for teams;

RECOMMENDATIONS

- Communicate with client about unification of current process for all teams providing information about benefits of using it;
- Engage other teams into process discussion;
- Include unit test coverage into Continuous Integration process;
- Update existed Continuous Integration process for all teams by adding a rule to fail a build in case of unit test code coverage is not met;
- Notify all stakeholders about new rules in Continuous process;
- Document this process in Confluence or on kb space

There is no quality gate based on Code Coverage

FACTS

- Code Coverage is calculated on CI but it doesn't act as Quality Gate

IMPACT

- There is a possibility that the unit test suite is not enough to cover all relevant test cases which lead to:
  - Missed defects
  - More complex refactoring
  - The less testable code in future

PROBLEM STATEMENT

Unit tests coverage is an important metric to understand how well code is covered with unit tests. This information is especially useful when complex conditional logic with many alternative flows is tested. Code coverage is measured not only to understand current status but to measure and control team progress. Ideal 100% coverage is not a goal because the exact threshold depends on many parameters like a technological stack, programming language, % of legacy code, etc.

Reviewing the code coverage result helps to identify code path(s) that are not covered by the tests. This information is important to improve the test collateral over time by reducing the testing debt.

RECOMMENDATIONS

- Agree with the team about code coverage for new code
- Setup code coverage quality gate on CI
  - Ideally, it should be a quality gate on NEW code only, not the whole code base
- Measure code coverage on CI Pull Request Verification job (See the previous slide)
- Use code coverage information during code review to identify gaps in unit test cases.

Absence of Mutation Testing

FACTS

- Current unit tests coverage is up to 80%
- Quality of unit tests is important for the team

IMPACT

- Unclear (opaque) unit tests quality and therefore possible lowered product quality

ADDITIONAL INFORMATION

Code coverage is used to understand problems in unit tests. Code coverage doesn't show how good the tests are; it shows how bad are they. For example if test coverage is 50%, it means that around 50% of functionality is not tested properly and even if all happy paths are covered, there're still a lot of alternative and negative scenarios, which could not work as intended. At the same time, if code coverage is 90%, it doesn't mean we are certainly safe, high coverage could be achieved by easy to write tests, which covers really nothing. High code coverage target may push developers to satisfy target instead of carefully test functionality.

RECOMMENDATIONS

- Introduce Mutation testing
  - Define if Mutation testing is applicable for the project
  - Choose tool for mutation testing (http://pitest.org/ is recommended)
  - Execute mutation tests for repositories with code coverage more that 80%
  - Analyze results and define if it is necessary improve unit testing approach for your teams
  - Create technical debt tasks to fix gaps in unit tests for most critical areas
  - Apply steps above on regular basis (per sprint ideally)
    - Consider incorporation of mutation tests in Delivery (CI\CD) Pipeline
- Reuse BRD-WFRM project experience in Mutation testing

Test-last approach is used for unit testing

FACTS

- The team uses unit tests as part of test automation strategy
- Unit test coverage services is higher than 90%:
  - Service A – 95.88%
  - Service B – 91.27%
- Tests are developed using test-last approach – tests are written when development of corresponded functionality is already complete

IMPACT

- Test-last approach for test automation in general and unit testing especially could lead to so-called defects of omission. Such kind of defects related to expected, but not implemented behavior of computer system. It happens due to the difficulty to code all possible behaviors without guidance of well-defined test cases or acceptance criteria. In such situation engineers tend to implement only “happy-path” and most trivial “negative/edge” behaviors.
- Even high level of test code coverage in case of test-last approach doesn’t guarantee quality as soon as **only written code is tested**, which can make this metric quite misleading.

RECOMMENDATIONS

- Consider switch to the test-first approach for unit testing, or Test-Driven Development
- Conduct TDD training and coaching

Coding Standards are not documented

FACTS

- There is no dedicated KB area for coding standards

IMPACT

- Lower Code Quality
- Lower Code Readability and Maintainability

ADDITIONAL INFORMATION

- Coding Standards are important to programmers for a number of reasons:
- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Standards improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

RECOMMENDATIONS

- Create separate KB page(s) to describe Coding Standards which usually describe:
  - Programming language usage (Exceptions, conditions, ternary operators, finalizers, etc.)
  - Architecture modules structure
  - Recommended structures usages
  - Third party components and libraries usage
  - The common use design patterns and anti-patterns
- Use the standards as a reference during code review process, and visa-versa all decisions made during conversation around source code is a subject to be added to Coding Standards.
- Ensure that Coding Standards are a part of a newcomer's guide

Coding Standards are not followed by the whole team

FACTS

- Green and Blue teams have coding standards and follow it
- Orange team does not follow coding standards

IMPACT

- Lower Code Quality
- Lower Code Readability and Maintainability
- Efficiency of existing coding standards is minimized because at least one third of the code are written without any standards

ADDITIONAL INFORMATION

- Coding Standards are important to programmers for a number of reasons:
- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Standards improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

RECOMMENDATIONS

- Convince Orange team to follow common coding standards and other engineering processes. As an option EngX Bootcamp training can be provided for customer teams.
- Create separate KB page(s) to describe Coding Standards which usually describe:
  - Programming language usage (Exceptions, conditions, ternary operators, finalizers, etc.)
  - Architecture modules structure
  - Recommended structures usages
  - Third party components and libraries usage
  - The common use design patterns and anti-patterns
- Use the standards as a reference during code review process, and visa-versa all decisions made during conversation around source code is a subject to be added to Coding Standards.
- Ensure that Coding Standards are a part of a newcomer's guide

There is no common project coding guidelines

FACTS

- Team refers to EngX Bootcamp program as a source of coding guidelines
- ReSharper is used for static code analysis to check violations of coding conventions
- There are no project specific coding guidelines that includes best coding practices and anti-patterns that might be used by team members for coding and code review.

IMPACT

- Lack of project specific guidelines may lead to inadequate learning curve when new team members are onboarded
- Improper indication of code issues (false-positives) due to different understanding of code requirements
- Possible rework in the Code Review process due to the review of the code that will not pass technical quality gates

PROBLEM STATEMENT

Code standards implies detailed knowledge base describing both general best practices and project particularities: project code structure, programming principles, programming rules of thumb, standard architecture approaches/(anti-)patterns.

Project specific coding standards simplify development process, facilitate best practices distribution across team members, reduce rework effort.

RECOMMENDATIONS

- Update existing project knowledge base adding the following articles:
  - Code review checklist (see example);
  - Review .NET development guideline and adapt for project needs if necessary;
  - Extend KB with project specific best practices and anti-patterns;
  - Prepare onboarding guideline for newcomers referring to prepared articles;
- Conduct series of Miniq sessions to get acquainted team members with updated KB;
- Revisit KB regularly, all decisions made during conversation around source code is a subject to be added to Coding Standards.

More details on Coding Standards & Guidelines:

1. Coding Standards & Guidelines at EPAM Microsoft CC;
2. .NET Guideline https://git.epam.com/EPMC-MSFT/net-guideline/blob/master/EPM-SPI_DotNETDevelopmentGuideline.md;
3. Code Review Guideline and Code Review Checklists at EPAM Java CC;
4. Recommended training courses:
   1. Clean Code for .Net;
   2. Series of “C# Design Patterns“ training courses at EPAM learning portal.

Not Formal Coding Standards

FACTS

- There is no dedicated document described coding standards, “do’s and dont’s” only

IMPACT

- Lower Code Readability
- Lower code quality due to technical errors
- Lower maintainability

RECOMMENDATIONS

- Create dedicated KB page(s) for coding standards for each technology
- Choose existing coding standards if any (for PL/SQL at least)
- Copy the chosen standards to KB and amend it in accordance with current informal agreements
- Update corresponding pages once any related question appears
- Use these coding standards as a reference for code review and newcomer onboarding

There is no static code analysis quality gates

FACTS

- There is no static code analysis

IMPACT

- Code quality degradation
- Code readability degradation
- More efforts during code review

RECOMMENDATIONS

- Introduce Static Code Analysis quality gates with SonarQube :
- Take standard SonarQube rule set
- Segregate **must not be** violated and could be violated rules.
- Setup Sonar quality gates in accordance with the agreement
- Break CI build in case of quality gates violation.

Static code analysis rules are not strict

FACTS

- Current Sonar quality gates assumes “A” rating for Security, Reliability and Maintainability Ratings
- Real “Technical Debt Ratio” is less than 1%

IMPACT

- Possible code quality degradation

PROBLEM STATEMENT

The Maintainability Rating scale can be stated by saying that if the outstanding remediation (Technical Debt ratio) cost is:

- <=5% of the time that has already gone into the application, the rating is A
- between 6 to 10% the rating is a B

In accordance with EngX experience Technical Debt ratio = 5% is to soft quality gate, even projects which do not care about code quality have similar values.

RECOMMENDATIONS

Introduce more strict quality gates against code smells:

- Take current SonarQube rule set
- Segregate **must not be** violated and **could be** violated rules.
- Setup Sonar quality gates in accordance with the agreement
- Break CI build in case of quality gates violation.

There is no static code analysis quality gates

FACTS

- SonarQube analysis is performed for information purposes only
- There is no static code analysis quality gates in delivery pipeline

IMPACT

- Code quality degradation
- Code readability degradation
- More efforts during code review

RECOMMENDATIONS

- Introduce Static Code Analysis quality gates :
- Take standard SonarQube rule set
- Segregate **must not be** violated and **could be** violated rules.
- Setup Sonar quality gates in accordance with the agreement
- Break CI build in case of quality gates violation.

Code Quality metrics are not used

FACTS

- ESLint is used as a quality gate
- There is an instance of SonarQube, but it is not used by the team due to technical limitations
- Code quality metrics are not calculated

ARTIFACTS

IMPACT

- High risk of code readability and maintainability degradation due to uncontrolled code complexity
- Code quality degradation due to less maintainable code
- More efforts during code review due to low code readability

RECOMMENDATIONS

- Introduce new version of SonarQube that can calculate code quality metrics
- Pay attention to the following metrics:
  - Class/Method cognitive complexity
  - File/method size
  - Method parameters count
  - Etc.
- Set up quality gates based on the metrics above:
  - Agree with the team on which metrics should be used and what thresholds shouldn't be exceeded
- Set up Sonar quality gates in accordance with the agreement:
  - Choose calculation of "New Period" approach. New Period could be started as a specific date when the new quality gates applied and updated after each release
  - Set up quality gates conditions based on "new code" metrics instead of Overall code metrics
- Break CI build in case of quality gates violations

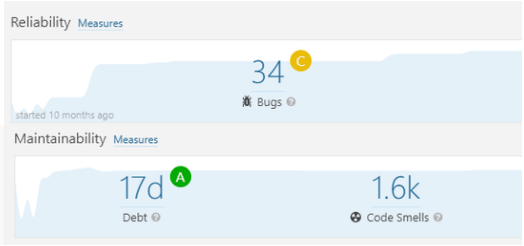
EngX

There is no static code analysis and code coverage quality gates

FACTS

- Linters do not break CI build;
- SonarQube is not incorporated into CI/CD pipeline;
- Static code analysis does not break builds on Pull Requests;
- Unit Test Code coverage target is not set

METRICS



IMPACT

- Code quality degradation
- Code readability degradation
- More efforts during code review
- Possible defect leakage

RECOMMENDATIONS

- Introduce SonarQube quality gates for **NEW** code:
- Set up Static Code Analysis quality gates:
  - Take standard SonarQube rule set;
  - Segregate must not be violated and could be violated rules;
- Set up Unit Tests Code coverage quality gates:
  - Agree with the team on code scope to be cover by unit tests;
  - Structure (if possible) code base for easy automatic identification of "to be covered" code;
  - Define minimum code coverage level (80%+ recommended);
- Set up Sonar quality gates in accordance with the agreement:
  - Choose calculation of "New Period" approach. New Period could be started as a specific date when the new quality gates applied and updated after each release
  - Set up quality gates conditions based on "new code" metrics instead of Overall code metrics
- Break CI build in case of quality gates violation;
- Break build on Development branch;
- Create build on Pull Request (to keep Development branch green) and break it as well;

There is no static code analysis and code coverage quality gates

FACTS

- Static code analysis does not break builds

IMPACT

- Code quality degradation
- Code readability degradation
- More efforts during code review

RECOMMENDATIONS

- Introduce SonarQube quality gates for **NEW** code:
- Static Code Analysis quality gates
  - Take standard SonarQube rule set
  - Segregate **must not be** violated and **could be** violated rules.
- Unit Tests Code coverage quality gates
  - Agree with the team what code should be covered by unit tests.
  - Structure (if possible) your code base for easy automatic identification of “to be covered” code
  - Agree with the team about minimum code coverage level for “to be covered code” (60%+ recommended)
- Setup Sonar quality gates in accordance with the agreement
  - Choose calculation of “New Period” approach. New Period could be started as a specific date when the new quality gates applied and updated after each release
  - Set up quality gates conditions based on “new code” metrics instead of Overall code metrics. See list of metrics here
- Break CI build in case of quality gates violation.
  - Break build on Dev branch
  - Create build on Pull Request (to keep Dev branch green) and break it as well

No static code analysis for C# part

FACTS

- There is no static code analysis for C# code

IMPACT

- Code quality degradation
- Code readability degradation
- More efforts during code review

RECOMMENDATIONS

Introduce Static Code Analysis quality gates with SonarQube :

- Take standard SonarQube rule set
- Segregate **must not be** violated and **could be** violated rules.
- Setup Sonar quality gates in accordance with the agreement
- Break CI build in case of quality gates violation.

In case of it is impossible to install SonarQube, some other tools could be used. See list in this post.

Static Code Analysis is not used

FACTS

- Static Code Analysis tools are not used

IMPACT

- Possible leakage of technical errors
- Additional time for code review due to spending time on potentially automated tasks
- Lower readability and maintainability due to missed issues during code review

RECOMMENDATIONS

- Choose appropriate static code analysis tool for each technology
  - For Oracle: consider SonarQube plugins (for Developer Edition only) or/and Toad capabilities
- Run Static code analysis tolls locally at least before each commit to fix all critical issues.
- Use static code analysis on CI as quality gate to be sure that there is no unaddressed violations

Flaws in Code Review process

FACTS

- Post-commit code review is used
- There is no dedicated tool for code review

IMPACT

- Lower code and product quality due to potentially missed defects;
- Inefficient code review process, hard to track that all issues are fixed, no statistics or evidences that code review is done;

RECOMMENDATIONS

- Use short-lived feature branches (less than 1 week) to segregate unverified (potentially wrong) code and verified (passed some checks like code review, automated testing, static code analysis)
- Pull Request functionality to conduct code review. Make sure that all review issues are fixed before Pull Request is merged.
- Use Coding Standards as a reference for code review issues (if applicable). Enrich code standards in case of new appropriate issue is found.

Absence of Code Review checklist (minor)

FACTS

- There is no code review checklist

IMPACT

- Lower code review quality
- Lower product quality

PROBLEM STATEMENT

Checklists are a highly recommended way to find the things you forget to do, and are useful for both authors and reviewers. Omissions are the hardest defects to find – after all, it's hard to review something that's not there. A checklist is the single best way to combat the problem, as it reminds the reviewer or author to take the time to look for something that might be missing. A checklist will remind authors and reviewers to confirm that all errors are handled, that function arguments are tested for invalid values, and that unit tests have been created.

Another useful concept is the personal checklist. Each person typically makes the same 15-20 mistakes. If you notice what your typical errors are, you can develop your own personal checklist (PSP, SEI, and CMMI recommend this practice too). Reviewers will do the work of determining your common mistakes. All you have to do is keep a short checklist of the common flaws in your work, particularly the things you forget to do. As soon as you start recording your defects in a checklist, you will start making fewer of them. The rules will be fresh in your mind and your error rate will drop. We've seen this happen over and over.

RECOMMENDATIONS

- Introduce common reviewers' checklist
  - Update the checklist based on leaked defects which could be found by code review
- Introduce author checklist at least for newcomers and junior developers

There is no pre-commit code review

FACTS

- Usually code review process is skipped in team

IMPACT

- Broken code is in repository;
- Other developers in team can be affected by bugs that were not found on time;
- Level of code quality is lower than can be;
- Code readability. There is different code style in team;

RECOMMENDATIONS

- Consider Code Review process as a part of CI during Pull/Merge requests;
- Introduce updated Code Review process to the team;
- Add Code Review Guidelines for the entire team (Coding Standards & Guidelines at EPAM Microsoft CC);
- Update existing project knowledge base adding the Code review checklist (see example);

Code Review process is not assisted by tools

FACTS

- Code Review is conducted on Merge Request
- Reviewers do not use any information from CI tools during code review

IMPACT

- Possible rework in the Code Review process due to the review of the code that will not pass technical quality gates
- Quality degradation due to missed defects during code review

PROBLEM STATEMENT

To conduct proper and thoughtful code review session, reviewer shouldn't be distracted to review source code aspect which could be performed by automated tools like naming conventions, formatting, .

One of the main goal of code review is to verify implementation of business logic and ensure that code follows best practices in terms of Clean Code and Clean Design rules. Thus it is very important to minimize routine review work and let reviewer to focus on human-only tasks.

RECOMMENDATIONS

- During code review take into account results of Merge Request verification job
  - Make a decision to start code review based on quality gates for static code analysis and code coverage
  - Check code coverage report to identify uncovered lines of code, it helps to verify completeness of unit test cases
  - Check list of “could be violated” violations to identify which of them should be fixed in this particular case

Unit Tests are not covered during code review

FACTS

- Usually Unit Tests are not covered by code review process

IMPACT

- Lower Unit Tests quality and therefore lower product quality and maintainability

RECOMMENDATIONS

- Treat unit tests as a first-class citizen:
  - Apply static code analysis rules to test code
  - Verify completeness and quality of unit tests during code review

Business logic is not reviewed during code review

FACTS

- In most cases, correctness of business logic implementation is not covered by code review process

IMPACT

- Lower product quality due to defects leaked from code review

SOME FACTS ABOUT CODE REVIEW PROCESS

- Latent defect discovery rate of formal inspection is in the 60-65% range. For informal inspection, the figure is less than 50%. The latent defect discovery rate for most forms of testing is about 30%
- Code review rates should be between 200 and 400 lines of code per hour
- Code review duration shouldn’t be more than 60 minutes

RECOMMENDATIONS

- Make sure that business logic is covered during code review at least for changes in critical functionality

Business logic is not reviewed during code review

FACTS

- In most cases, correctness of business logic implementation is not covered by code review process

IMPACT

- Lower product quality due to defects leaked from code review

SOME FACTS ABOUT CODE REVIEW PROCESS

- Latent defect discovery rate of formal inspection is in the 60-65% range. For informal inspection, the figure is less than 50%. The latent defect discovery rate for most forms of testing is about 30%
- Code review rates should be between 200 and 400 lines of code per hour
- Code review duration shouldn’t be more than 60 minutes

RECOMMENDATIONS

- Make sure that business logic is covered during code review at least for changes in critical functionality

Secure Development

Secure Development practices are not established

FACTS

- Team didn't pass secure development training
- Secure development expertise is available in a team
- There is no formal Secure SDLC approach on the project

IMPACT

- High risk of user data leakage;
- Late detection of flaws/vulnerabilities in the system;
- Cost increase as a result of late detection and resolution of issues with major or blocker severity;
- Overall increase of intrinsic business risks for the organization

STATEMENT

A Secure SDLC process ensures that security assurance activities such as penetration testing, code review, and architecture analysis are an integral part of the development effort.

Security by design one of the secure engineering principles. Solution Architecture and Code should be developed in accordance with standard practices aimed to protect the solution from identified threats

RECOMMENDATIONS

Revise security principles to be implemented in accordance to EPAM Security CC recommendations:

- "S-SDLC as a part of EngX": https://kb.epam.com/display/EPMCSEC/Security+as+a+part+of+EngX
- Officially published Secure Development guideline based on the first link: https://pal.epam.com/pal_method_plugin/guidances/whitepapers/resources/EPM-SPI_SecureDevelopmentGuideline.docx
- Available Security Training:
  - https://kb.epam.com/display/EPMCSEC/Team+On-Boarding+compilation
  - https://kb.epam.com/display/EPMCSEC/External+Security+Courses

Secure Development practices are not established

FACTS

- There is no formal Secure SDLC approach on the project;
- Using Black Duck is not included delivery pipeline
- Security metrics are not collected by the team

IMPACT

- User data leakage;
- Late detection of flaws/vulnerabilities in the system;
- Cost increase as a result of late detection and resolution of issues with major or blocker severity;
- Overall increase of intrinsic business risks for the organization

PROBLEM STATEMENT

A Secure SDLC process ensures that security assurance activities such as penetration testing, code review, and architecture analysis are an integral part of the development effort.

Security by design one of the secure engineering principles. Solution Architecture and Code should be developed in accordance with standard practices aimed to protect the solution from identified threats

RECOMMENDATIONS

Overall recommendation:

- Pass internal Epam security audit;
- Create Security plan;
- Establish Quality Gates for code security quality ;
- Develop Implementation or Migration plan in relation to security compliance goal (can be a part of Project Management Plan)
- Develop Security testing plan or include security section in Test plan;
- Discuss with WK possibility to include Black Duck usage in development pipeline;
- Perform DAST AND SAST types of testing. Document test results, including any unexpected variations discovered during testing.
- Conduct regular vulnerability analysis (Vulnerability assessment reports) using Black Duck (WK)

Lack of knowledge in security development process

FACTS

- There is no formal Secure SDLC approach on the project;
- Using Veracode is not included delivery pipeline
- Defects from Veracode are not reported into Bug Tracker System
- Security metrics are not collected by the team

IMPACT

- User data leakage;
- Late detection of flaws/vulnerabilities in the system;
- Cost increase as a result of late detection and resolution of issues with major or blocker severity;
- Overall increase of intrinsic business risks for the organization

PROBLEM STATEMENT

A Secure SDLC process ensures that security assurance activities such as penetration testing, code review, and architecture analysis are an integral part of the development effort.

Security by design one of the secure engineering principles. Solution Architecture and Code should be developed in accordance with standard practices aimed to protect the solution from identified threats

RECOMMENDATIONS

Overall recommendation: revise implemented security principles in accordance to EPAM Security CC recommendations:

- Courses:
  - https://learn.epam.com/detailsPage?id=fa3c908c-63df-489c-b8b3-f42775cff49c
  - https://learn.epam.com/detailsPage?id=e78e6e48-0dc0-45ae-a30b-fb9565cf6aa8
  - https://kb.epam.com/display/EPMCSEC/Team+On-Boarding+compilation
- Articles:
  - "S-SDLC as a part of EngX":

    https://kb.epam.com/display/EPMCSEC/Security+as+a+part+of+EngX
  - https://kb.epam.com/display/EPMITSVCFAQ/Secure+Software+Development+LifeCycle
  - https://pal.epam.com/pal_method_plugin/guidances/whitepapers/resources/EPM-SPI_SecureDevelopmentGuideline.docx

Lack of Secure Development knowledge

FACTS

- Team `members did not pass Secure SDLC trainings
- There is no automated security testing

IMPACT

- Possible security issues which can lead to reputational and financial losses

RECOMMENDATIONS

- Revise security principles to be implemented in accordance to EPAM Security CC recommendations:
  - “S-SDLC as a part of EngX”: https://kb.epam.com/display/EPMCSEC/Security+as+a+part+of+EngX
  - Officially published Secure Development guideline based on the first link: https://pal.epam.com/pal_method_plugin/guidances/whitepapers/resources/EPM-SPI_SecureDevelopmentGuideline.docx
- Available Security Training:
  - https://kb.epam.com/display/EPMCSEC/Team+On-Boarding+compilation
  - https://kb.epam.com/display/EPMCSEC/External+Security+Courses

Different team members evaluate security practice differently

FACTS

- At initial completion of the EngX Survey team members (including team leads) ranked most of the questions from the “Secure Development” section as Low-to-Medium level, however Architect’s evaluation is High level

IMPACT

- If team members, especially Team leads, are not aware about Security practices embedded into the Solution Architecture, they won’t be able to effectively implement those practices; most likely some of the security guards will be unintentionally violated

RECOMMENDATIONS

- For Architect, to conduct a knowledge transfer session to the team, to present Security principles used on the project
- Ensure Security is documented in the project Knowledge Base and included into the onboarding guide for new team members
- Overall recommendation: revise implemented security principles in accordance to EPAM Security CC recommendations:
  - “S-SDLC as a part of EngX”: https://kb.epam.com/display/EPMCSEC/Security+as+a+part+of+EngX
  - Officially published Secure Development guideline based on the first link: https://pal.epam.com/pal_method_plugin/guidances/whitepapers/resources/EPM-SPI_SecureDevelopmentGuideline.docx
  - Available Security Training: https://kb.epam.com/display/EPMCSEC/Team+On-Boarding+compilation

DevOps

Binary Repository is not used

FACTS

- Binary repository (e.g. Artifactory, Nexus, etc.) is not used
- Each deployment (including Prod) includes artifacts rebuilding process

IMPACT

- Lack of deployment flexibility and agility;
- Possible hardly reproducible production defects due to artifacts rebuild (technically artifacts deployed on production are different from artifacts tested on other environments)

PROBLEM STATEMENT

By having a dedicated storage for a build binaries and artifacts you can easily deploy an environment of any required release or a build version. This enables you to track down bugs that were reported for a specific product version. Sometimes it can be very critical.

You can either use some specific network location as an artifacts and binaries storage or you can rely on a special tool for providing this capability, e.g. Artifactory or Apache Archiva. The special tools will introduce additional flexibility and capabilities for you project.

RECOMMENDATIONS

- Introduce binary repositories like Artifactory or Nexus;
- Change deployment process
  - Publish built Docker images to Docker Registry provided by binary repository
  - Deploy Docker images from Docker Registry
  - Prepare separate Ci job to deploy any image from repository
- Change release strategy to follow Build Binaries Only Once rule

Long Unstable build (Parking team)

FACTS

- E Build duration is more than 1 hour
- Integration tests which are maintained by separate team are included in the build
- Stability of integration tests are low

IMPACT

- Huge inefficiency or work stoppage in case of failed build

RECOMMENDATIONS

- Exclude Integration tests from the build process, run integration tests as optional post build step
- Make it possible to deploy the binaries on Dev environment without integration tests execution

Absence of SAST/DAST tools

FACTS

- Application is not checked by SAST/DAST tools

IMPACT

- NHigh risk of security breach

ADDITIONAL INFORMATION

SIt's estimated that 90 percent of security incidents result from attackers exploiting vulnerabilities at the application level.

https://www.csoonline.com/article/2978858/is-poor-software-development-the-biggest-cyber-threat.html

RECOMMENDATIONS

- Employ SAST/DAST tools in the project (https://pal.epam.com/pal_method_plugin/guidances/whitepapers/resources/EPM-SPI_SecureDevelopmentGuideline.docx)
  - Choose and integrate SAST tool in CI/CD pipeline (https://kb.epam.com/pages/viewpage.action?pageId=729091381)
  - Choose and integrate DAST tool in CI/CD pipeline (https://kb.epam.com/pages/viewpage.action?pageId=729092012)

Absence of monitoring/alerting tools

FACTS

- Application state is not monitored

IMPACT

- Possibly undetected service downtime
- Slow Incident response
- No performance metrics tracking

RECOMMENDATIONS

- Review existing application logs to determine if they are suitable to build alert monitoring on
- Change/Add log messages if needed
- Configure existing log aggregation platform (Splunk, ELK) to email alerts
- Start using APM tools (e.g., AppDynamics, Elastic APM)

Absence of secrets management

FACTS

- Application secrets is located along with source code in human readable format

IMPACT

- Possible secrets leakage
- Unauthorized access to resources
- Data leakage
- Possible security issues which can lead to reputational and financial losses

PROBLEM STATEMENT

- Holding credentials in human readable format is unsecure. That might lead to major security incidents caused by stolen credentials and unauthorized access to internal systems (f.e database).

RECOMMENDATIONS

- Revise security principles to be implemented in accordance to EPAM Security CC recommendations:
  - “S-SDLC as a part of EngX”: https://kb.epam.com/display/EPMCSEC/Security+as+a+part+of+EngX
  - Officially published Secure Development guideline based on the first link: https://pal.epam.com/pal_method_plugin/guidances/whitepapers/resources/EPM-SPI_SecureDevelopmentGuideline.docx
  - Available Security Training:
    - https://kb.epam.com/display/EPMCSEC/Team+On-Boarding+compilation
    - https://kb.epam.com/display/EPMCSEC/External+Security+Courses

Absence of configuration management

FACTS

- All application configurations is located along with source code
- App configuration is versioned along with app source code

IMPACT

- Impossible to change application configuration without releasing new version
- Impossible to deploy existing version to a new environment
- Cost and risk of changing the configuration is higher

PROBLEM STATEMENT

Application configuration should be externalized, otherwise it's the same as hardcode configuration in the source code. For example, for throubleshoot purposes need to increase logging level, when app configuration is located along with source code without releasing new version it's imposibble. Best practice is to externalize app configuration to allow modify it without changing source code.

RECOMMENDATIONS

- Externalize application configuration to Spring Cloud config server

Automated tests are not an essential part of delivery pipeline

FACTS

- Automated regression/smoke suites are not included in CI/CD pipeline
- Developers do not know about any QA automated tests

IMPACT

- Lower Test Automation efficiency due to long feedback cycle from automated tests

RECOMMENDATIONS

- Include execution of Smoke Automated tests (if any) into CI/CD pipeline after each deployment
- Include execution of Regression Automated tests (if any) into CI/CD at least on daily basis

CI pipelines and jobs are not stored in VCS

FACTS

- Improvements for storing CI pipelines in VCS are in progress;

IMPACT

- Not clear access point for all external dependencies;
- Existing vulnerabilities of well-known public packages are not defined;

PROBLEM STATEMENT

Build only once on commit. It prevents unexpected issues related with compile/build process. Build results of the same commit could be different on different environments because of unexpected changes in environments itself (SDK version, build scripts changes, changes in dependencies, etc.). Thus once binary artifact is built it should be used in further deployments instead of rebuilding from source codes. It means that CI pipeline should have step with coping build artifacts to dedicated repository. It could be shared folder or special binary repository like Nexus/Artifactory (preferable).

RECOMMENDATIONS

- The following items are in progress:
  - Design strategy how and when to initiate binary build based on the changes and merges in the VCS and the result of the review process. Decide when to propagate the build and to which environment.
  - Choose and setup binary storage tool;
- Implement the strategy in CI/CD pipeline

There are not enough environments

FACTS

- Code from feature branches are not deployed on separate environments;
- Development team has not enough environments to preform code validation on early stage;

IMPACT

- Late feedback on developed functionality due to awaiting time when code merged into master branch;
- Defects can be missed with other code changes after merging with other code;
- Cost of defects are increased significantly;

RECOMMENDATIONS

- Discuss the opportunity to set up separate environments for development/manual/automation testing with client in order to perform testing activities from feature branches (environments for feature testing);
- Set up access policies to new environments for the team (security policy);
- Set up the process of getting feedback on quality from feature branches on recently developed functionality;

Production/runtime environment requirements are not specified

FACTS

- High and low environments have a different configurations;

IMPACT

- Long feedback loop,
- Difficult for investigation;
- Risk of delivery delays as soon as released software can't be used in production and needs urgent rework;

PROBLEM STATEMENT

Clear vision of production environment and infrastructure setup is necessary to create effective development and QA process. The rule of thumb is to minimize difference between all environments are used for software development process that prevents issues when the same application build works in one environment but doesn't work in the next environment due to the environment specific issues.

RECOMMENDATIONS

- Define requirements to production environment and infrastructure based on architecture significant requirements and quality attributes;
- Review current Development and QA environments and spot the differences based on application configuration, data volume, high availability set up, components interconnections, etc.
- Adjust Development and QA environments to replicate Production environments completely or at least to be similar with predictable, manageable differences. Such differences must be covered with additional set of smoke tests (auto or manual);

Manual Deployment

FACTS

- All deployments are performed manually

IMPACT

- Production failures due manual error-prone deployment process;
- Automated deployment is an enabler for CI/CD and test automation practices;
- Time waste for manual deployment process and fixing possible deployment issues

RECOMMENDATIONS

- Choose a way how to deploy each module automatically
  - For database deployment consider Toad capabilities or tools like LiquiBase, Flyway, etc.
  - For other technologies consider using of scripting languages like PowerShell, Bash, or configuration management tools like Ansible, Puppet, etc.
- Implement corresponding deployment approach
- Create CI job to deploy new versions
- Get rid of manual deployment, deploy new versions only via CI

Absence of CI Server

FACTS

- There is no CI server and CI process

IMPACT

- It is impossible to efficiently apply Continuous Integration principles without CI server
- There is no convenient way to run any task automatically

RECOMMENDATIONS

- Choose CI tool
  - Jenkins is recommended
  - Also consider CI module incorporated into git repository
- As a first step, implement static code analysis, test automation and deployment jobs and run on demand.
- Next, run corresponding jobs automatically by schedule or triggered by git events (push to branch, merge request creation, etc.).

There is no automate deployment on high/low environments

FACTS

- There is a manual deployment process for Staging and Production environments;

IMPACT

- Release frequency is lower than it can be;
- Delay in getting feedback (late code delivery to QA team);
- Human factor: incorrect environment can be deployed on;
- Not everyone in the team can deploy applications (core and mobile);
- Manual deployment process is often a time-consuming and thankless task;
- Spend time on configuration of each new environment;

RECOMMENDATIONS

- Create CD pipeline to automate deployments to various environments;
- Version all builds that are being stored in artefact repository and use it build promotion;

DB deployment is not automated

FACTS

- There is a manual DB deployment process in the project;
- DB deployment takes 10-15 minutes;

IMPACT

- Waste team time on safety and control;
- Unstable Builds and delay with Deployments and Release;
- Team should support data for Production and Nonproduction databases;
- PII disclosing;
- Incorrect environment can be deployed on;

PROBLEM STATEMENT

Bad database deployments can be costly to recover from and possibly fatal to the project.

RECOMMENDATIONS

- Set up Database deployment process and discuss it with customer:
  - Select tools for DB deployment process;
  - Set up pre-deployment analysis of potential performance issues;
  - Configure Deployment process (create DB backups, apply proposed changes in DB and etc);
  - Verify that the final state of new DB model meets expectations;
- Define database deploy challenges that can be faced during setting up the procedure (for instance, persisting data during upgrades and etc);
- Create a roll back plan in case of failed deployment;
- Document process on confluence or kb for new commers;

Permissions are not configured to run deployment on STAGE/PROD

FACTS

- As of now Developers are able to run a job deploying to STAGE/PROD environment

IMPACT

- PROD/STAGE could be broken by accident deployment initiated by improper person

PROBLEM STATEMENT

Only authorized staff should have permissions to run deployment job on high environment

RECOMMENDATIONS

- Configure permissions in Jenkins properly

Late Feedback From Test Automation

FACTS

- Automated Smoke Tests and E2E tests executed once per 8 hours on Master branch
- Execution of full autotests suite is limited by server's capacity

IMPACT

- Lower team performance due to late defects identification
- Lower efficiency of autotests because it is not an essential part of development pipeline

ADDITIONAL INFORMATION

Contract testing is a way to ensure that services (such as an API provider and a client) can communicate with each other. Without contract testing, the only way to know that services can communicate is by using expensive and brittle integration tests.

RECOMMENDATIONS

- Consider introducing "Contract Testing" practice to improve mutual contract understanding and identify breaking changes on early stages
  - https://docs.pact.io/ - Pact is a contract testing tool
  - Verifying Microservice Integrations with Contract Testing Video about contract testing
- Shift critical part of automated tests on Pull Request verification job
  - Identify environment which can be used to run automated tests on Pull Request
    - In case of necessity create new environment
  - Choose the most important cases from "Automated Smoke Tests" suite and "E2E API Tests for the service" (see Feedback Diagram) to be executed on Pull Request.
  - Total duration of those tests shouldn't exceed 30 minutes.
  - To reach that duration consider tests parallelization.
  - Implement corresponding steps in Pull Request Verification jobs.

Low Test Automation Coverage

FACTS

- Current Test Automation Coverage is about 77%
- Current Defect Containment Efficiency is about 70%
- Stability of Autotest is about 90%

IMPACT

- Lower product quality
- Lower team performance due to late defects identification

RECOMMENDATIONS

- Improve Test Automation Coverage
  - Prove (or refute) the hypothesis that low DCE is caused by Low Test Automation coverage.
  - Define test case areas which should be automated first.
  - Prepare plan for automatization of that areas and approve it with the customer.
- Improve tests stability
  - Perform root cause analysis for failed tests and identify most common type of failures
  - Prepare corrective actions to eliminate that failures
  - Track Automation stability metric and target it to be more then 95%

Absence of Mutation Testing

FACTS

- Current unit tests coverage is up to 90%
- Quality of unit tests is important for team and customer

IMPACT

- Unclear (opaque) unit tests quality and therefore possible lowered product quality

ADDITIONAL INFORMATION

Code coverage is used to understand problems in unit tests. Code coverage doesn't show how good the tests are; it shows how bad are they. For example if test coverage is 50%, it means that around 50% of functionality is not tested properly and even if all happy paths are covered, there're still a lot of alternative and negative scenarios, which could not work as intended. At the same time, if code coverage is 90%, it doesn't mean we are certainly safe, high coverage could be achieved by easy to write tests, which covers really nothing. High code coverage target may push developers to satisfy target instead of carefully test functionality.

RECOMMENDATIONS

- Introduce Mutation testing
  - Define if Mutation testing is applicable for the project
  - Choose tool for mutation testing (http://pitest.org/ is recommended)
  - Execute mutation tests for repositories with code coverage more that 80%
  - Analyze results and define if it is necessary improve unit testing approach for your teams
  - Create technical debt tasks to fix gaps in unit tests for most critical areas
  - Apply steps above on regular basis (per sprint ideally)
    - Consider incorporation of mutation tests in Delivery (CI\CD) Pipeline

Lack of Automated Quality Gates on Pull Request

FACTS

- To approve Pull Request Team Lead manually check Sonar code analysis statistic and code coverage

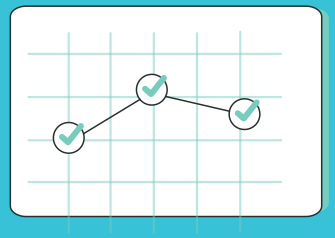
IMPACT

- Possible code quality violations due to human factor

RECOMMENDATIONS

- Fail CI Build for Pull Request in case of lack of Code Coverage for new code and any Critical+ static code analysis issues

Quality Assurance



Test process is inefficient

FACTS

- Integration functionality is not covered with E2E automated tests
- Common Perfect Systems testing strategy assumes no manual testing involvement
- Manual QA was added into South Star team 1.5 month ago
- Unbalanced team: 1 QA engineer/10+ developers
- Developers are responsible for automated tests creation process
- Manual QA receives development changes on Pre-Production environment
- Only Architect, Dev Lead and BA participate in Three Amigo Sessions to prepare and refine requirements

IMPACT

- Cost of fixing defect is increased significantly due to lack of QA capacity
- High risk of late feedback on regression functionality
- Defects are hidden under other functionalities that were not tested
- Spending development time on clarifications for QA requests and as a result delay with a delivery

RECOMMENDATIONS

- Define responsibility area for manual QA engineers
- Adjust current test strategy in order to incorporate automated E2E tests
- Conduct workshops to improve team understanding on testing scope and strategy
- Involve QA team into SDLC process on early stage, including them as participants in 3 amigos sessions and reviewing development test cases
- Using metrics, analyze and adjust existed “No-QA” testing approach:
  - Introduce balanced test pyramid
  - Consider adding more automation E2E tests
- Consider substitution of manual QA engineer with a full stack (manual + automated skills) automation engineer

Test case management process is inefficient

FACTS

- Excel is used for storing manual test cases
- Regression plans are stored in Excel sheets
- Jira is a team Requirement Management Tool, but it's not integrated with Excel
- There is no test traceability matrix

IMPACT

- Long learning curve: newcomers adaptation takes time
- There is no transparency and traceability for development team and all stakeholders on testing results

RECOMMENDATIONS

- Consider using a common tool in order to store information on how testing is to be done, plan testing activities and report the status of quality assurance activities
- Conduct comparison of several TM tools using best practices
- Define the functionality that can be moved to the Test Management Tool: Test Case/Test Plans/Release test plans
- Agree with other teams to use Test Management Tool in collaboration with Jira
- Prepare materials and conduct training sessions for QA team
- Move new releases' test plans into selected test management tool
- Consider adding Traceability matrix in order to get understanding about test coverage in a team

Metrics-driven QA monitoring and reporting process is not established

FACTS

- There is no monitoring and reporting process for manual QA engineer
- There is no unified approach to collect QA metrics in a team
- QA metrics are not used as a tool to improve QA processes
- Action items/improvements are not defined based on metrics results

IMPACT

- A moment when test strategy becomes inefficient can be missed and as a result product quality degradation can happen
- Without quality metrics it is impossible to measure testing effectiveness, development process efficiency and define areas for improvements
- Difficult to plan improvements and testing activities in sprints and releases
- Cost of fixing defects are increased due to inefficient testing process

RECOMMENDATIONS

- Define correct stakeholders and set of metrics that they can be interested in across team
- Track and analyze at least the following QA metrics on regular basis:
  - Defect Containment Effectiveness
  - Reopen Defect ratio
  - Invalid Defect ratio
- Once E2E automated testing is implemented, collect the following metrics:
  - Test Automation Stability
  - Test Automation Coverage
  - Test Automation Progress
  - Automation tests execution time
  - ROI (save time of manual functional software engineers)
- Consider having unified dashboard and report template for a QA team

Metrics are not collected and analyzed on regular basis on the project

FACTS

- QA metrics are not used as a tool to improve QA processes
- Action items/improvements are not defined based on metrics result

IMPACT

- Unknown quality of testing strategy and therefore product quality could be better
- There is no transparency and traceability for Development team and customer on outcome testing results
- Quality Metrics allow to measure testing effectiveness, development process efficiency and define areas for improvements
- Difficult to plan improvements and testing activities in future sprints and releases;
- Cost of fixing defects are increased due to inefficient testing process

RECOMMENDATIONS

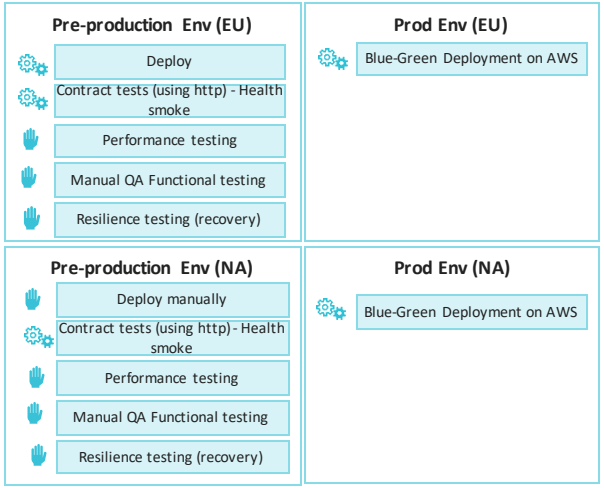
- Consider to connect EPAM PERF tool (https://perf.epam.com) as metrics dashboard;
- Define correct stakeholders and set of metrics that they can be interested in;
- Track and analyze at least the following QA metrics on regular basis:
  - Defect Containment Effectiveness
  - Reopen Defect ratio
  - Invalid Defect ratio
- Consider **EPAM Report Portal** (https://rp.epam.com) usage to increase transparency of AQA teamwork collecting following metrics:
  - Test Automation Stability
  - Test Automation Coverage
  - Test Automation Progress
  - Automation tests execution time
  - ROI (save time of manual functional software engineers)

End Users do not verify the product before production release

FACTS

There is no UAT activity on any environments

ARTIFACT



IMPACT

- Risk of misunderstanding between end users and development team
- Risk of defect being identified on production are increased
- High cost of defect fixing
- Lack of end user education

RECOMMENDATIONS

- Agree with end users UAT process
- Create UAT business scenarios based on the acceptance criteria set
- Ensure that real end users execute the UAT business scenarios
- Conduct regular meetings to communicate status and address questions as they come up
- Schedule time for defect resolution and re-testing of existed functionality
- Prioritize defects and changes as they are identified

Lack of testing environments

FACTS

- There are Feature, Pre-Production and Production environments only
- Testing is done on Pre-Production environment

IMPACT

Potential conflicts between on-going testing and investigation of defects/incidents from production

RECOMMENDATIONS

- Setup separate environment integrated with all third-party components for QA activities:
  - Manual new feature testing
  - Manual regressions testing
  - Automated “development” testing
  - Automated E2E tests
  - Performance testing
- Establish a process to anonymized data transfer from higher environment to QA environment

Dev Team does not have clarity on QA processes

FACTS

- QA processes are not transparent to Dev Teams

IMPACT

- Lower product quality due to inefficient collaboration between Dev and QA teams

RECOMMENDATIONS

- Use Test Strategy document as a tool to understand QA processes and initiate discussion about efficient Dev-QA collaboration
- Consider creation of mixed Scrum teams with EPAM Devs and LTI QAs

Too late QA involvement

FACTS

- QA receive development changes once per week on Staging environment
- Customer verification first on Dev Environment

IMPACT

- Cost of defect fixing is increased significantly
- Late feedback on regression functionality
- Because of deploy 1 per week, delay with Defect and Feature acceptance testing results
- Defects are hidden under other functionality
- Spending time on clarifications for QA requests
- Lack of information about last minute changes

RECOMMENDATIONS

- Align development, QA and customer to conduct software testing immediately after feature implementation or defect fixing;
- Discuss the process of quality assurance with the client and formalize test strategy;
- Involve QA team into SDLC process on early stage:
  - 3 amigos approach for clarify requirements https://www.agilealliance.org/glossary/three-amigos/);

Test Management Tool is not used

FACTS

- Google doc is used for storing test cases;
- Regression plans are storing in google doc tabs;
- Jira is used as a Requirement Management Tool;

IMPACT

- Newcomers adaptation takes time;
- QA team productivity is lower than it can be;
- There is no transparency and traceability for Development team and customer on outcome testing results;

PROBLEM STATEMENT

Test management tool is a software which may be applied to management of any test activities over the entire development lifecycle. It helps to store information on how testing is to be done, plan testing activities and report the status of quality assurance activities.

RECOMMENDATIONS

- Define the functionality that can be moved to the Test Management Tool:
  - Test Case/Test Plans/Release test plans;
- Agree with customer to use Test Management Tool (Jira + QA Space);
- Select the TMT using EPAM best practices https://kb.epam.com/display/ETSTCC/How+to+select+tool
- Conduct training sessions for QA team and customer;
- Move new releases' test plans into selected TMT

Developers are not involved in Quality Engineering

FACTS

- Developers are not involved in Quality Engineering processes. There is lack of visibility and understanding of test strategy;

IMPACT

- Development team spends significant effort on QA team consultancy.

RECOMMENDATIONS

- Introduce EPAM Quality Engineer in QA squad to ensure connectivity between EPAM development and customer's QA teams.

Too long feedback loop from integration testing

FACTS

- Change Requests to implemented features can be raised in several months after the feature implementation
- There is no immediate integration testing phase after feature completion

IMPACT

- Delays in release schedule due to unexpected CRs
- Lower team performance in CR implementation due to changes in old “forgotten” functionality

RECOMMENDATIONS

- Align feature development and integration testing between different teams (including customer teams) to conduct integration testing immediately after feature implementation
- Introduce “Contract Testing” practice to improve mutual contract understanding and identify breaking changes on early stages

Performance testing is not included in delivery pipeline

FACTS

- Team does NOT conduct performance testing as on-going process on team neither on the solution level;
- Performance testing environment does not exist;

IMPACT

- Application availability, scalability and resilience;
- Time to market due to preparation and execution a set of performance tests and environment;

RECOMMENDATIONS

- Prepare environment for Performance testing or plan to do it before the release;
- Prepare tool evaluation for performance testing process;
- Prepare E2E test scenarios on feature and solution level;
- Non-functional testing should be automated and included in CD pipeline at least on Sprint basis;

Testing is not finished in sprint

FACTS

- New feature testing can be not finished in the sprint by EPAM QAs
- Full release and regression testing is performed by another vendor during the dedicated test phase

IMPACT

- Later feedback about code quality leads to extra time for fixing and changes
- Release cycle can't be accelerated with the dedicated testing phase

RECOMMENDATIONS

- Allocate enough test resources for new feature testing in the sprint for all development streams (EPAM and MediaKind)
- Strengthen Definition of Done with criteria that testing is done, all defects are fixed and closed to complete sprint and stories
- Automate regression testing and non-functional requirements testing and execute them in the sprint

Not all important metrics are calculated

FACTS

- TA Stability and coverage is tracked, Invalid Defect Ratio is tracked, Bug Growth and Created vs Resolved trend is tracked
- Defect Density, Defect Containment, Reopen defects ratio are not tracked

IMPACT

- Quality Metrics allow to measure testing effectiveness, development process efficiency and define areas for improvements

RECOMMENDATIONS

- Once a common test strategy is defined, set up quality metrics measurements to evaluate the effectiveness of test strategy (DCE)
  - Adjust bug reporting with additional fields to be able to identify internal defects and corresponded external defects to calculate DCE for each release (Affected Version, environment, testing types, etc.)
  - Set up a process to track regression defects and defects found by test automation
  - Create dashboards for metrics
  - Perform root cause analysis to identify defects leakage reasons, the testing type that should have found and test team that related
- For now, to measure testing effectiveness we may perform root cause of defects reported by other vendor and external defects from client to identify issues that missed in new feature testing and possibly adjust the testing approach

QA Metrics are not used

FACTS

- QA metrics are not regularly used as a tool to improve QA processes

IMPACT

- Unknown quality of QA processes and therefore product quality could be better

RECOMMENDATIONS

- Consider to connect EPAM PERF tool as metrics dashboard
- Track and analyze at least the following metrics on regular basis:
  - Defect Containment Effectiveness
  - Reopen Defect ratio
  - Invalid Defect ratio
  - TA stability (consider EPAM Report Portal usage)

There is no test automation process

FACTS

- There is no common test automation process and strategy on the project;
- Regression testing takes more one week;
- Test Automation process is planned but not implemented

IMPACT

- Late feedback about code quality that leads to possible delays with delivery;
- Late delivery due to time on regression testing;
- QA team demotivation, performing routine work;

RECOMMENDATIONS

- Set up test automation process on the project;
- Define and prioritize Scope for automation testing;
- Evaluate Test Automation tool for the project;
- Automate Smoke tests firstly;
- Collaborate with developers regarding test automation coverage to minimize duplications
  - For each feature define tests that should be covered on different levels - unit, API and UI;
- Create a traceability matrix for tracking requirements;
- Increase test automation coverage to execute regression testing automatically in sprints;
- Update Test Plan and Test Strategy adding Test Automation process there;

Absence of Automated Tests (Unit, Functional, Etc.)

FACTS

- There is no automated testing (unit, integration, UI);

IMPACT

- Lower product quality due to missed regression defects;
- Inefficient QA strategy due to waste for manual run and verification of test cases

RECOMMENDATIONS

- Choose a tool for automated tests for each technology
  - Consider automated functional testing modules in Toad or utPLSQL or similar tools for PL/SQL code.
- Consider to use UI automation for End-To-End testing
  - Consider Selenium tool as UI test engine
  - Consider Cucumber as BDD tool for writing tests scenarios

Developers are not involved in test automation

FACTS

- Automated Tests are written by dedicated team member

IMPACT

- Lower testable product; if test automation is not a part of developers' responsibilities, they tend to write more less testable code (breakable selectors, poor API design, etc.)
- High Test coverage duplication; Test Automation Engineer doesn't know what logic is covered by unit and integration tests so some parts of logic (code) are unnecessary covered by automated tests as well
- Lower product quality; Collaboration between Test Automation Engineer and Developers are not efficient because they are not highly interested in each other activities, therefore quality of test suites is lower than could be.

RECOMMENDATIONS

- Educate developers with basics of test automation
- Educate developers with basics of current test automation framework and tools
  - Developers should be able to understand root cause of test failure at least
- Share responsibility for fixing failed tests between Test Automation Engineer and Developers
- As a next step, introduce practice to automate acceptance criteria (or critical part of them) by feature authors (developers) as a part of user story development

Smoke automation tests are not included in CI/CD pipeline

FACTS

- There are no Smoke tests (nor API nether UI) included in CI/CD pipeline
- Short time for integration testing on Staging environments before release

IMPACT

- Late feedback about code quality that leads to possible delays with delivery;
- Late delivery due to time on execution regression testing;
- Defects can be missed if tests are not executed manually on automation testers local machines

RECOMMENDATIONS

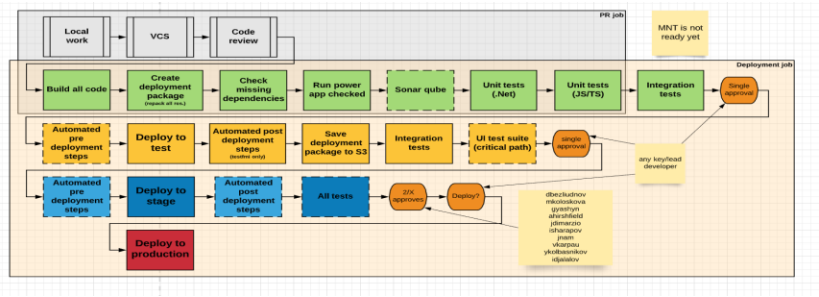
- Include Smoke automation tests into CI/CD pipeline:
  - Use headless mode for UI tests;
  - Use mocks for API tests if potential changes are expected;
- Define and set right priorities for automation testing team;
- Collaborate with developers regarding test automation coverage to minimize duplications
  - For each feature define tests that should be covered on different levels - unit, API and UI;
- Create a traceability matrix for tracking requirements;

Smoke test suite is not used by developers and absents in CI/CD pipeline

FACTS

- Automated tests are **not** valuable for development team
- Smoke suite (critical path) is not a part of CI/CD pipeline
- AQA process is not transparent to development teams

ARTIFACTS



IMPACT

- Lower Test Automation efficiency due to long feedback cycle from automated tests
- There is no transparency and traceability for Development team and Customer on outcome automation testing results;
- Late feedback about code quality that leads to possible delays with delivery;

RECOMMENDATIONS

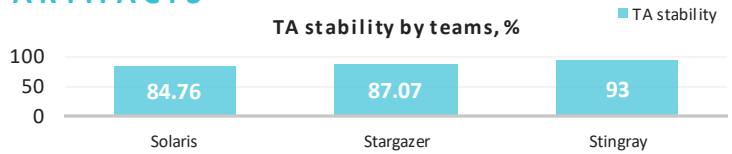
- Define a process how to make automated tests more valuable for development teams in order to get feedback as early as it possible;
- Prepare a critical path suite (should not take more than 15 minutes to execute) that contains independent tests and share it with development teams;
- Create guide how to configure and run automated test on a local development environment and document it on a Confluence/KB/Newcomer guide;
- Check results critical path execution before code review process;
- Include Critical Path tests into CI/CD process in order to have feedback as soon as Deployment process is done;

There are low test automation coverage and stability automated tests

FACTS

- Automated tests are executed more than **10 hours**
- **Low test automation** coverage in Solaris team (28%)
- TA stability:
  - Solaris: 1% failed, but from 210 tests are executed only 182
  - Stargazer: 13% (43 tests on 09/04) failed tests
  - Stingray: 7% failed tests

ARTIFACTS



IMPACT

- Lower product quality due to inefficient collaboration between Development and AQA teams
- Additional time waste for RCA of failed tests
- Automation team spend time on stabilization of automated tests
- Failed results of automated tests if they unstable can be ignored due to lack of tests
- Cost of fixing defects are increased due to inefficient testing process

RECOMMENDATIONS

- Define set of most business valuable and stable tests
- Include selected in **p.1** tests into CD process
- **Define set of** most unstable tests and create a strategy how to fix them
- Split automation tests by priority (P1, P2, P3, P4) or by type (smoke or regression) if not split yet;
- Consider preparing dedicated “unstable test suite” to have other tests stable and reliable
- Discuss and set up a target for “TA Stability” metric. Stability of UI tests should be more than **95%.** Stability for other type of tests (API, Integration) should be **98%+**
- Consider possibility to shorten automation tests duration using :
  - Parallelization
  - Changes in test design/architecture (API tests)

EngX

Automated E2E tests are not introduced

FACTS

- E2E automated tests missed in a process
- No manual QA approach was used in team
- Manual tests are not considered to be automated
- Developers don’t review manual test cases

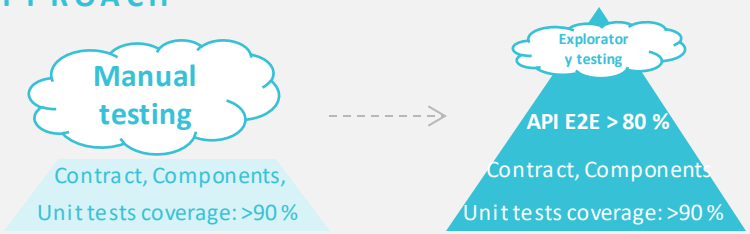
IMPACT

- Late feedback about E2E tests results that leads to possible delays with delivery
- Increasing QA effort on regression testing
- Possible late delivery due to time on regression testing
- Defect omission due to lack of manual QA capacity
- Over time there is team demotivation, performing routine work

RECOMMENDATIONS

- Define and prioritize scope for E2E automation testing
- Evaluate test automation tool for the project and create test automation framework
- Define a process how to make E2E automated tests more valuable for development teams in order to get feedback as early as possible
- Prepare a critical path suite from manual test cases created by manual QA engineer
- Include critical path tests into CI/CD for each MR in order to have feedback as soon as deployment process is done
- Consider implementing acceptance test driven approach

APPROACH



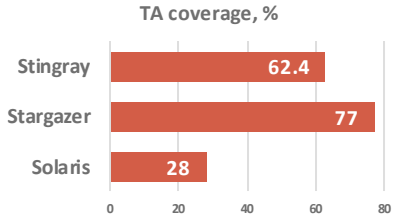
EngX

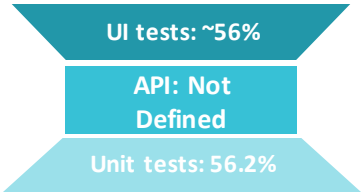
Test automation pyramid is not used

FACTS

- Unit test coverage in **Front End solution is low**!
- Business logic is covered by UI tests only
- There is a lack of API (not integration) tests

METRICS





IMPACT

- Time for conducting RCA can be increased due to lack of understanding on a layer where a defect's place is;
- End-to-end test's coverage has higher maintenance cost;
- Testing is not scalable;
- Potential tech debt increasing;
- Cost of fixing defects are increased due to inefficient testing process;

RECOMMENDATIONS

- Define an approach to shift business logic validation on lower layer (from UI to API) and **discuss it with a client**;
- Define and prioritize Scope for API automation testing;
- Evaluate Test Automation tool for API testing on the project;
- Reconsider test automation approach to execute as much as possible on API level;
- Increase API test automation coverage to execute regression testing automatically in sprints;
- Update Test Plan and Test Strategy adding API tests in Test Automation process;

There is no dedicated environment for automation testing process

FACTS

- Test Automation scripts are executing on the same environment with manual testing;

IMPACT

- Potential conflicts between on-going manual and automated testing;
- Difficulties to results of runs and to reproduce failed results in case of including into CI process;
- No opportunity to use it for tests included in CI;

RECOMMENDATIONS

- Discuss the opportunity to separate environments for manual/automation and performance testing with client;
- Define environment requirements for Automation testing;
- Define data creation mechanism and an obfuscation approach for automation test scenarios;
- Configure a dedicated environment for automation testing process;

Low stability of autotests

FACTS

- About 10% of autotests are failed each run

IMPACT

- Team tends to ignore results of autotests if they unstable
- Additional time waste for RCA of failed tests

RECOMMENDATIONS

- Review your testing pyramid to be sure that your test automation strategy minimizes amount of unstable flaky tests
- Introduce **EPAM Report Portal** (https://rp.epam.com) to track and analyze results of each tests run.
- Define set of most unstable tests and create a strategy how to fix them.
- Consider to prepare dedicated “unstable test suite” to have other tests stable and reliable
- Set up a target for “TA Stability” metric. Stability of UI tests should be more than 95%. Stability for other type of tests (API, Integration) should be 98%+

Automated tests are not an essential part of delivery pipeline

FACTS

- Automated tests are not included in CI/CD pipeline
- Developers are not involved in implementation of automated tests

IMPACT

- Lower Test Automation efficiency due to long feedback cycle from automated tests

RECOMMENDATIONS

- Include execution of Smoke Automated tests into CI/CD pipeline after each deployment
- Involve developers into implementation of automated tests which cover at least most critical acceptance criteria
  - Test Automation Engineer should cover all other cases
  - Shift left autotests execution to Merge Request job

Automated Tests are not executed on Pull Request

FACTS

- Earliest time automated tests executed on DEV-INT Environment

IMPACT

- Long feedback cycle from automated tests make them almost useless for developers
- Lower efficiency of test automation defects fixing.

RECOMMENDATIONS

- Introduce suite of automated tests which should be run on Pull Request. The suite should :
  - Cover most critical parts of the system
  - Duration less that 20 minutes
  - Stability more that 95% (99%+ is preferable)
- Integrate execution of that suite into Pull Request CI job

Low Test Automation Coverage

FACTS

- Test Automation coverage is 36% which requires manual regression testing to be performed before release
- No common test automation strategy, the current solution is focused only on features that under development and testing by EPAM

IMPACT

- Release cycle can't be accelerated with extensive manual testing and dedicated testing phase
- Late feedback about code quality that leads to possible delays with delivery

RECOMMENDATIONS

- Increase test automation coverage to execute regression testing automatically in sprints
- Collaborate with developers regarding test automation coverage to minimize duplications
  - For each feature define tests that should be covered on different levels - unit, API and UI and who will implement what tests (developer or test automation engineer)

Test Automation Coverage is low

FACTS

- There is 27.7 % of all test cases are covered with automation testing
- Approximate execution time 1 h for regression pack;

IMPACT

- Late feedback about code quality that leads to possible delays with delivery;
- Release cycle can't be accelerated with extensive manual testing and dedicated testing phase
- Late delivery due to time on execution regression testing;
- Execution time: Long time to understand root cause of failures in for flaky tests

RECOMMENDATIONS

- Increase test automation coverage to execute regression testing automatically in sprints;
- Collaborate with developers regarding test automation coverage to minimize duplications;
- Create a traceability matrix for tracking requirements;
- Consider possibility to shorten automation tests duration using
  - Parallelization
  - Changes in test design/architecture

Time-consuming automated tests (minor)

FACTS

- Autotests duration is 40 minutes
- Autotests stability is about 85% (15% flaky tests)

IMPACT

- Long feedback cycle which makes them almost useless for developers
- Long time to understand root cause of failures in for flaky tests

RECOMMENDATIONS

- Consider possibility to shorten autotests duration using
  - Parallelization
  - Changes in test design/architecture
- Investigate most common root causes for flaky tests
  - Prepare mitigation strategy for most often failures
  - Introduce Report Portal as a tool to manage test results and track flaky tests

Automated API tests take significant time (Gatekeepers only)

FACTS

- Full set of automation tests run consequently on 6 repositories;
- Execution of full set of automation tests takes about 1 hour
- Automation tests are not split by priorities (Smoke and Regression);
- Full automation regression testing is performed on demand;
- No acceptance testing on the feature branches;
- Parallel test execution was considered before and decided not to go this way;

IMPACT

- Late feedback on basic functionality
- Time to complete deployment is higher than it can be (not save manual testing time);

RECOMMENDATIONS

- Define common automation test strategy more thoroughly that includes common test automation approach on deferent levels:
  - Define Test Automation strategy in common (TAA used on the project);
  - Automation acceptance testing should be run in sprints on feature branch/pull requests;
  - Re-consider run automation tests in parallel on QA and Stg environments (by tests or by repositories);
  - Instead of running full set of automation tests on demand, include P1 or Smoke tests in deployment process and P2-P4 or regression type of tests as a chain build after deployment that can be executed on demand as well;
- Split automation tests by priority (P1, P2, P3, P4) or by type (smoke or regression);
- Define code review rules for automation engineers (policy)

Defect management process is not structured

FACTS

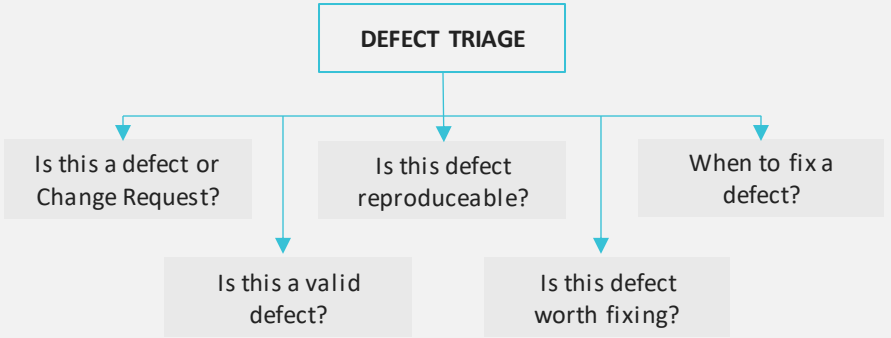
- Priority and Severity fields are not used for defect's description
- Defects are not segregated from change requests
- Developers report defects using

IMPACT

- Handling regression defects requires a lot of rework
- Slowed delivery also becomes a roadblock for software tester to write effective test cases
- Possible increasing the maintenance cost
- Possible increased duration of bug fixing cycle

RECOMMENDATIONS

- Unify defects tracking process for all team members and document it on kb/Confluence space
- Introduce Severity and Priority fields in defects:
  - Severity should reflect impact on the product
  - Priority should reflect real urgency of bug fixing
- Defects should be reported using the following rules:
  - Contain detailed description in order to have common understanding;
  - Priority and severity are set up according to rules
- Establish weekly Defect Triage Meeting with Product Owners:



EngX

Regression defects are not tracked by all teams

FACTS

- There is no defined process to track regression defects for all teams
- Bug Triage process exists but not documented and not structured
- Not all teams analyze test strategy effectiveness

ARTIFACTS

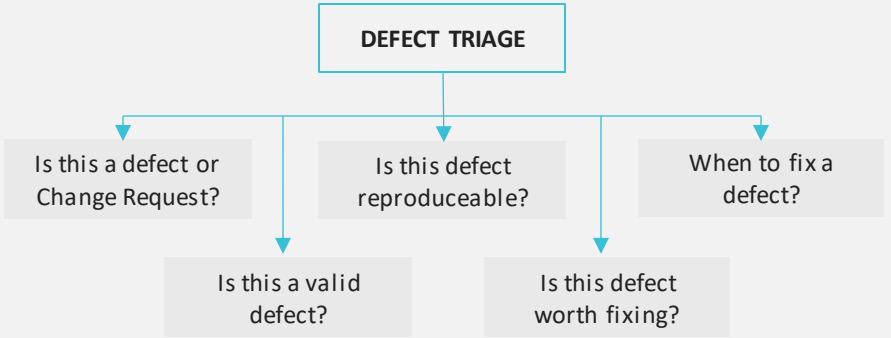
Current process:

IMPACT

- Handling regression defects requires a lot of rework
- Regression defects can prove to be a lot challenging when the deadline draws closer
- Slowed delivery also becomes a roadblock for software tester to write effective test cases
- Possible increasing the maintenance cost

RECOMMENDATIONS

- Unify Regression defects tracking process;
- Establish weekly Defect Triage Meeting with Product Owners;
- Be ready to each meeting :
  - Have enough information to understand the defect
  - Have reported under correct project and category
  - Have mentioned the priority and severity of the defects
  - All the detailed information provided in the defect to understand it correctly to all the participants



EngX

Defects are not segregated from change requests

FACTS

- Defects and change requests are the same entity

IMPACT

- It is impossible (too time consuming) to collect and analyze defects and CR related metrics, so it becomes much harder to improve QA and BA processes

RECOMMENDATIONS

- Consider using of dedicated JIRA issue type for Change Requests
- As less recommended approach use custom fields or labels to identify Change Requests
- Establish process to analyze what defects are CRs and transform such defects to CRs
- Exclude CRs from calculation of all defects related metrics

Severity field is not used

FACTS

- Only priority field is used in bugs report

IMPACT

- Less efficient defect management process
- Possible increased duration of bug fixing cycle

PROBLEM STATEMENT

Both Severity and Priority are attributes of a defect and should be provided in the bug report. This information is used to determine how quickly a bug should be fixed.

Severity defines the impact that a given defect has on the system, so how bad the defect is.

Priority is defined as the order in which a defect should be fixed in accordance to business value or

RECOMMENDATIONS

- Introduce Severity field in defects.
- Severity should reflect impact on the product.
- Priority should reflect real urgency of bug fixing.

Root cause analysis of leaked defects is not performed on regular basis

FACTS

- There is no root cause analysis process on the project for Staging and Production environments;

IMPACT

- QA team productivity is lower than it can be;
- There is no transparency and traceability for Development team and customer on outcome testing results;
- Possible similar defects are missed on early stage;

PROBLEM STATEMENT

Root cause analysis is an approach for identifying the underlying causes of an issues in a product so that the most effective solutions can be identified and implemented.

Performing RCA in high quality leads to increasing project quality.

RECOMMENDATIONS

- Perform RCA on low level environment on daily basis in order to catch defects early as possible;
- The RCA process for each defect reported from Staging and Production environments should be defined;
- Describe the reasons why defects were leaked to Staging or Production environment, e.g. no enough time to complete testing, regression, environment, data specific, missed in requirements;
- Create a test case for missed defect, automate it and include into regression pack to validate;
- Establish a causal graph between the root cause and the problem;
- Test Lead should create action items to avoid this situation in future;

High rate of invalid defects

FACTS

- Invalid defect ratio is 39% when the industry target is 10%

IMPACT

- Invalid defects are waste for development and test teams because of the required time for reporting, reproducing, triage, but in fact no bring any value

RECOMMENDATIONS

- Breakdown invalid defects by different resolutions
- Define the most common reasons for invalid defects (environment, not clear business logic, etc.)
- Define actions to eliminate these reasons and minimize invalid defect ratio

Test Strategy is not up to date

FACTS

- The Test Strategy is documented;
- The documented Test Strategy is not up-to-date;

IMPACT

- Lack of communication of the test approach to the various stakeholders;
- Quality aspects are not prioritized according to business priority;
- Quality assurance risks are not analyzed, prioritized and mitigated;

PROBLEM STATEMENT

Test Strategy is a plan for defining the testing approach, and it answers to questions like what you want to get done and how you are going to accomplish it. It is a most important document for any QA team in software testing.

The purpose of updating a test strategy is to plot the strategic plan on how test efforts will be distributed across the project at the current project stage.

RECOMMENDATIONS

- Define responsible people for deciding to approve changes in existed Test Strategy ;
- Formalize test strategy in accordance with well-known (EPAM guideline is recommended: Test Strategy Guidelines)
  - If necessary, pass educational materials Effective Test Strategy webinar
- Discuss Test Strategy with all concerned parties and make amendments if any
- Constantly review and adjust test strategy to achieve better product quality

There is no common tests strategy across all testers

FACTS

- EPAM QA performs only new feature testing in sprints and regression testing based on the capacity of the test team
- Full release and regression testing is performed by another vendor during the dedicated test phase
- Testing is not aligned between all participants, there is no common test strategy

IMPACT

- Work duplication because both test teams do the same testing
- Possible product quality risks because of absence common test strategy

RECOMMENDATIONS

- Define common test strategy that includes all testing types and common approach for testing between all participants:
  - Instead of having dedicated test team for all scrum dev teams, allocate testers for each scrum team to perform new feature testing in a sprint
  - Regression testing should be automated and run in sprints
  - Non-functional testing should be automated and run in sprint
- Set up regular communication and test results sharing to avoid work duplication
- (optional) Involve QA Manager to manage all testing activities

Non-Functional Requirements are not defined

FACTS

- Non-Functional requirements are not defined yet

IMPACT

- Possible critical production issues due to failed NFR

RECOMMENDATIONS

- Formalize Non-Functional Requirements
- Introduce NFR testing in accordance with defined requirements
- Set NFR testing results as quality gate