Best Appliance Repair Experts

Created By: Mehreen Akmal, Jenn Bushey

The Problem

Best Appliances is a large home appliance chain that has numerous retail stores throughout Canada. When customers buy appliances and other electronic equipment, they can choose to purchase a support plan. Customer-facing appliance experts will then come to the customer's residence to fix problems with the device.

Things have not been good with the Best Appliances and if something isn't done soon, the company will be forced to abandon this very lucrative business line and fire all the employees, including your architect team.

The Task

You are tasked to come up with a new architecture to improve the reliability of the application. The current ticket system is a large monolithic application that was developed years ago.
Customers are complaining the appliance specialist is never showing up due to lost tickets.
Often the wrong appliance specialist shows up to fix something they know nothing about. Customers have been complaining that the system is not always available for web-based problem ticket entry.
Change is difficult and risky in this large monolith - whenever a change is made, it takes too long and something else usually breaks (the product owner calls new feature deployments "Bug releases"). Due to reliability issues, the monolithic system frequently fails - they think it's' mostly due to an increase in usage and the number of customers using the system, but they're not sure.

The Current System

1. Best Appliances repair experts are added and maintained in the system through an administrator, who enters their local, availability and skills.
2. Customers who have purchased the support plan can enter a problem ticket using the Best Appliance website. Customer registration for the support services is part of the system. The system bills the customer on an annual basis when their support period ends by charging their registered credit card.
3. Once a trouble ticket is entered in the system, the system then determines which appliance expert would be the best fit for the job based on skills, current location, service, area, and availability (free or currently on a job).
4. The Best Appliance repair expert is notified via a text message that they have a new ticket. Once this happens an email or SMS message is sent to the customer (based on their profile preference) that the expert is on the way.
5. The Best Appliance repair expert uses a customer mobile application on their phone to access the ticketing system to retrieve the ticket information and location. The repair expert can also access a knowledge base through the mobile app to find out what things have been done in the past to fix the problem.
6. Once the repair expert fixes the problem, they mark the ticket as "complete", The repair expert can then add information about the problem and add the resolution to the knowledge base.
7. After the system receives a notification that the ticket is complete, the system sends an email to the customer with a link to a survey which the customer then can reply to.

Additional Info

- Ticket assignment is a complicated process, and the algorithms change frequently to fine-tune the process.
- Once a ticket is assigned to a repair expert, it is immediately routed to that expert. If it can't be routed, it is immediately reassigned.
- Unit-of-work transactions are very important due to workflow issues with the current system.
- Ticket creation is customer-facing and must be scalable and available for ticket entry.

Functional Requirements

	Architectural Characteristics
Repair Expert Management • Administrator manages repair expert's information (availability, knowledge, skills). • Assign ticket to best fit expert or reassign if not available using latest algorithm.	Performance, Fault Tolerance, Modularity, Scalability, Simplicity
Customer Access – Web • Customer registers in support plan. • Customer enters a ticket in web application.	Elasticity, Scalability, Fault Tolerance, Reliability Modularity, Testability, Performance, Deployability
Database • Stores registered customers credit card information. • Stores ticket and history of ticket with "completed" status.	Fault Tolerance, Modularity, Scalability, Security
Payment System • Charges registered customer on an annual basis at the end of membership.	Performance, Security, Modularity, Reliability, Scalability, Fault Tolerance, Deployability
Communication • Notify expert of assigned ticket (SMS). • Notify customer about expert visit (email or SMS). • Email customer survey and notification of completed ticket.	Scalability, Reliability, Modularity, Security, Elasticity Fault Tolerance, Testability, Performance
Repair Expert Access - Mobile Application • Have access to query current ticket information and historical tickets. • Update ticket status to "complete, add problem details and resolution.	Elasticity, Scalability, Fault Tolerance, Modularity, Testability, Performance, Reliability, Deployability

Non-Functional Requirements

	Architectural Characteristics
• Increasing customers therefore system should use horizontal scaling (add servers with a lead strategy).	Scalability
• Able to handle surges in user activity during peak times.	Elasticity
• Able to continue normal operation despite the presence of system or hardware failure. • Backup tickets daily to ensure no loss of tickets. • System should apply timestamp to every ticket update to ensure the history of changes and the most recent change is recoverable. • Ability to handle errors while running if the internet connection goes down or if there's a power outage or hardware failure.	Fault Tolerance
• Able to add new functionality with ease to upgrade system when scaling up.	Modularity
• Deploy components in rolling deployment to ensure functioning system and reducing downtime.	Testability

Assumptions

These considerations should guide the design and development of the system to meet Best Appliance’s requirements effectively.

- Prioritize Quality and Budget Over Time
- Zero Downtime
- Avoid Confusing Customers During Rollout
- No Immediate Expansion Plans

Prioritize Quality and Budget Over Time

This means that the focus should be on delivering a high-quality system within the allocated budget, even if it means compromising on the delivery timeline.

This approach requires careful planning and resource management to ensure that the system meets quality standards without exceeding the budget. It may involve prioritizing critical features and implementing them thoroughly, while non-essential features may be deferred to future releases.

Zero Downtime

The system must be available at all times and cannot afford any periods of downtime, even during maintenance or upgrades.

This requirement necessitates the use of strategies such as rolling updates, blue-green deployments, or canary releases to ensure continuous availability. It also requires robust monitoring and alerting mechanisms to quickly identify and resolve issues that could lead to downtime.

Avoid Customer Confusion

Best Appliances wants to ensure a smooth transition for customers during the rollout phase, especially when introducing a new user interface.

This implies that the rollout should be carefully planned, with considerations for user training, communication, and support. It may involve a phased rollout approach, where the new user interface is gradually introduced to different segments of customers to minimize confusion and disruptions.

No Plans for Immediate Expansion

There are no indications that Best Appliances plans to expand its business beyond its current functionality.

While the system should be designed to accommodate future growth, there is no immediate need to build features or capabilities that are not directly related to the current business model. This allows for a more focused development effort and avoids unnecessary complexity.

Trade-Off Analysis

- Fault Tolerance vs. Cost
- Scalability vs. Complexity
- Modularity vs. Performance

Trade-Off Analysis

Counting the number of times each characteristic was mentioned in the business requirements, we find our most desired characteristics are Fault Tolerance, Scalability, and Modularity.

Count of Characteristics from Business Requirements		7	7	7	5	4	4	4	3	1
Characteristics Architecture Style	**Cost**	**Fault Tolerance**	**Scalability**	**Modularity**	**Performance**	**Elasticity**	**Reliability**	**Testability**	**Deployability**	**Simplicity**
Layered	5	1	1	1	2	1	3	2	1	5
Pipeline	5	1	1	1	2	1	3	3	2	5
MicroKernel	5	1	1	3	3	1	3	3	3	4
Service Based	4	4	3	4	3	2	4	4	4	3
Event Driven	3	5	5	4	5	3	3	2	3	1
Space Based	2	3	5	3	5	5	4	1	3	1
Microservices	1	4	5	5	2	5	4	4	4	1
Orchestration-Driven	1	3	4	3	2	3	2	1	1	1

Event Driven, Space Based, or Microservices

Based on the business requirements, we found that Event Driven, Space Based, or Microservices architecture styles fit the best with our desired characteristics.

Service Granularity

Examining disintegrators and integrators help determine the service granularity of the software architecture. Disintegrators are factors that tend to drive components or services apart, while integrators are factors that tend to bring components or services together.

Disintegrators

Domains - Customer Management, Repair Expert Management, Ticket System:
Separating these domains allows for modularization, ease of maintenance, and better scalability. This enables independent development and upgrades for each domain.

Scalability and Fault Tolerance:
The greatest disintegrator factor is scalability and fault tolerance, as the business is expected to grow in the number of stores and users while maintaining high availability and reliability.
This requires implementing strategies such as horizontal scaling, fault-tolerant design, and continuous monitoring to ensure the system can handle increased load and remain available under various conditions.

Disintegrators

Security:
Customer profile and payment information require separate security measures to keep credit card information encrypted. This involves regularly assessing and updating security measures.
Implementing strong encryption, secure storage practices, and regular security audits are essential to protect sensitive customer data.

Extensibility:
There is no indication that current services will be expanding in functionality, indicating low impact to granularity.
While the system should be designed to accommodate future changes and growth, the focus can be on meeting current business needs without overengineering for potential future requirements. This allows for a more focused and efficient development effort.

Integrators

Workflow and Choreography:
The Ticket System comprises sub-domains for ticket creation, assignment, routing, and completion. These sub-services need to communicate with one another to manage the ticketing process effectively.

Database Transactions/Relationships:
The system follows the BASE (Basically Available, Soft state, Eventually consistent) model for database consistency. This approach allows database relationships to be more flexible and broken apart, prioritizing system availability (24/7) and fault tolerance over immediate consistency.

Shared Code:
There is no shared code among the identified domains. Each domain works independently, contributing to the overall modularity and maintainability of the system.

Coupling and Contracts

Maintain a flexible and modular system that can adapt to changing requirements and scale effectively.

Examining coupling and contracts can help in determining the preferred software architecture by providing insights into how components interact and depend on each other.

Coupling

Loose Coupling between Domains:
Loose coupling between microservices allows for independence in development and maintenance. Changes in one domain are less likely to impact others, promoting modularity and flexibility.

Loose Data Coupling:
Allows for changes in the database structure without affecting other components that use the data.

Moderate Communication Coupling:
Necessary for functionality but minimize direct dependencies between components. Using well-defined interfaces and avoiding tight coupling ensures that changes in one component do not adversely affect others.

Contracts

Domain Contracts:
Well-defined contracts specify the responsibilities and interfaces of each domain. This ensures that each domain adheres to its designated role and provides a clear understanding of how they interact.

Customer Management to Payment System:
A strict contract is necessary to maintain semantic coupling since information has type-specific constraints.

Web Application to Customer Management/Ticket System:
A strict contract is required as certain fields will need to be entered with a specific format, maintaining the contract format.

Data Contracts:
Strict data contracts ensure that changes in the database structure do not break other components relying on that data. This is crucial for maintaining data consistency across the system.

Microservices

By mapping the business requirements to the architecture characteristics and performing trade-off analysis we determined the best fitting architecture styles. Granularity disintegrators/integrators, contracts, and coupling to determine the optimal architecture style for Best Appliances.

Architecture Diagram - Context

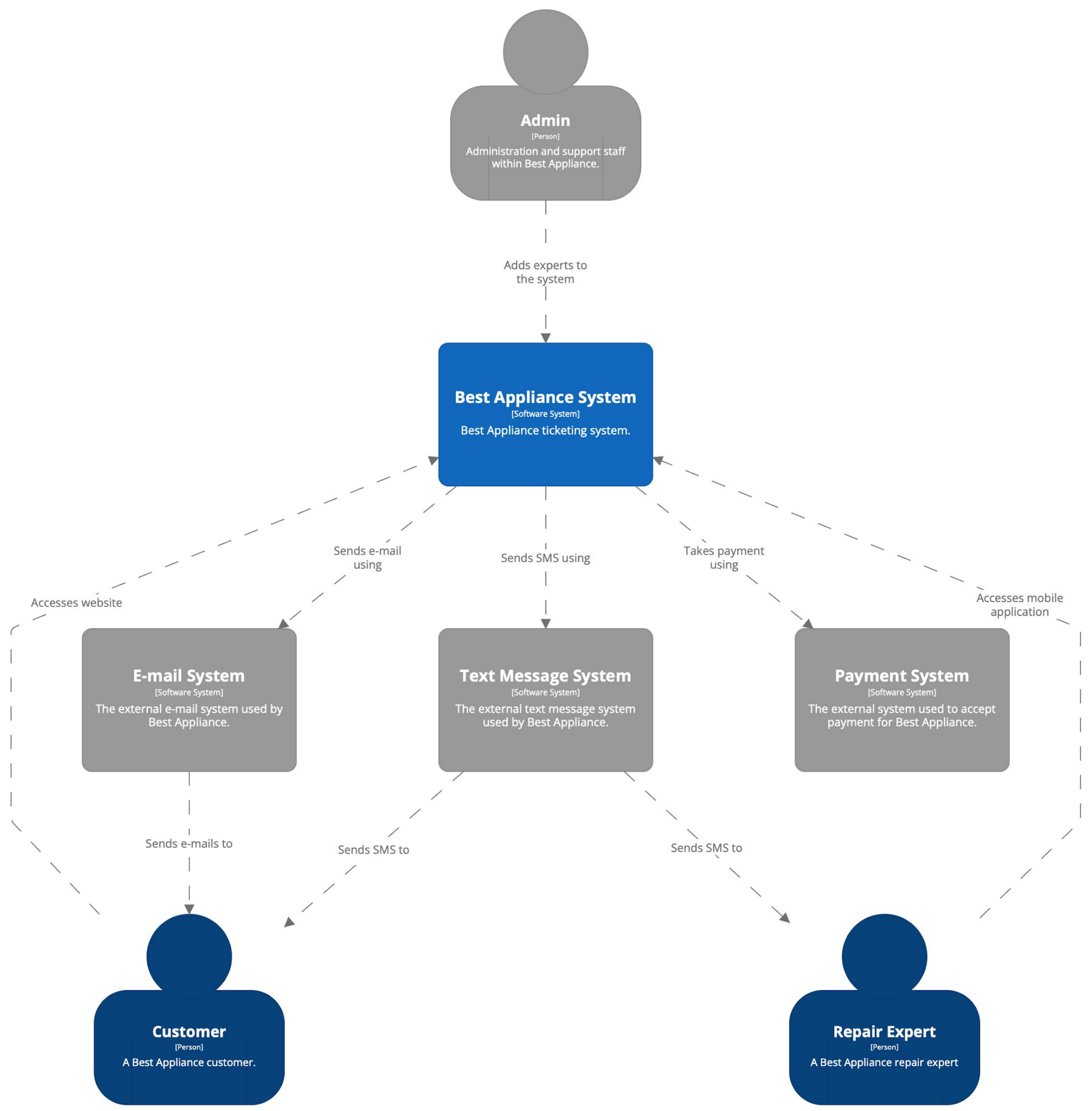
Best Appliances System interacts with:

Stakeholders:

- Customer
- Repair Expert
- System Administration

External Systems:

- External Email System
- External Text Message System
- External Payment System


Admin
[Person]
Administration and support staff within Best Appliance.
Adds experts to the system
Best Appliance System
[Software System]
Best Appliance ticketing system.
Accesses website
Sends e-mail using
Sends SMS using
Takes payment using
Accesses mobile application
E-mail System
[Software System]
The external e-mail system used by Best Appliance.
Text Message System
[Software System]
The external text message system used by Best Appliance.
Payment System
[Software System]
The external system used to accept payment for Best Appliance.
Sends e-mails to
Sends SMS to
Sends SMS to
Customer
[Person]
A Best Appliance customer.
Repair Expert
[Person]
A Best Appliance repair expert

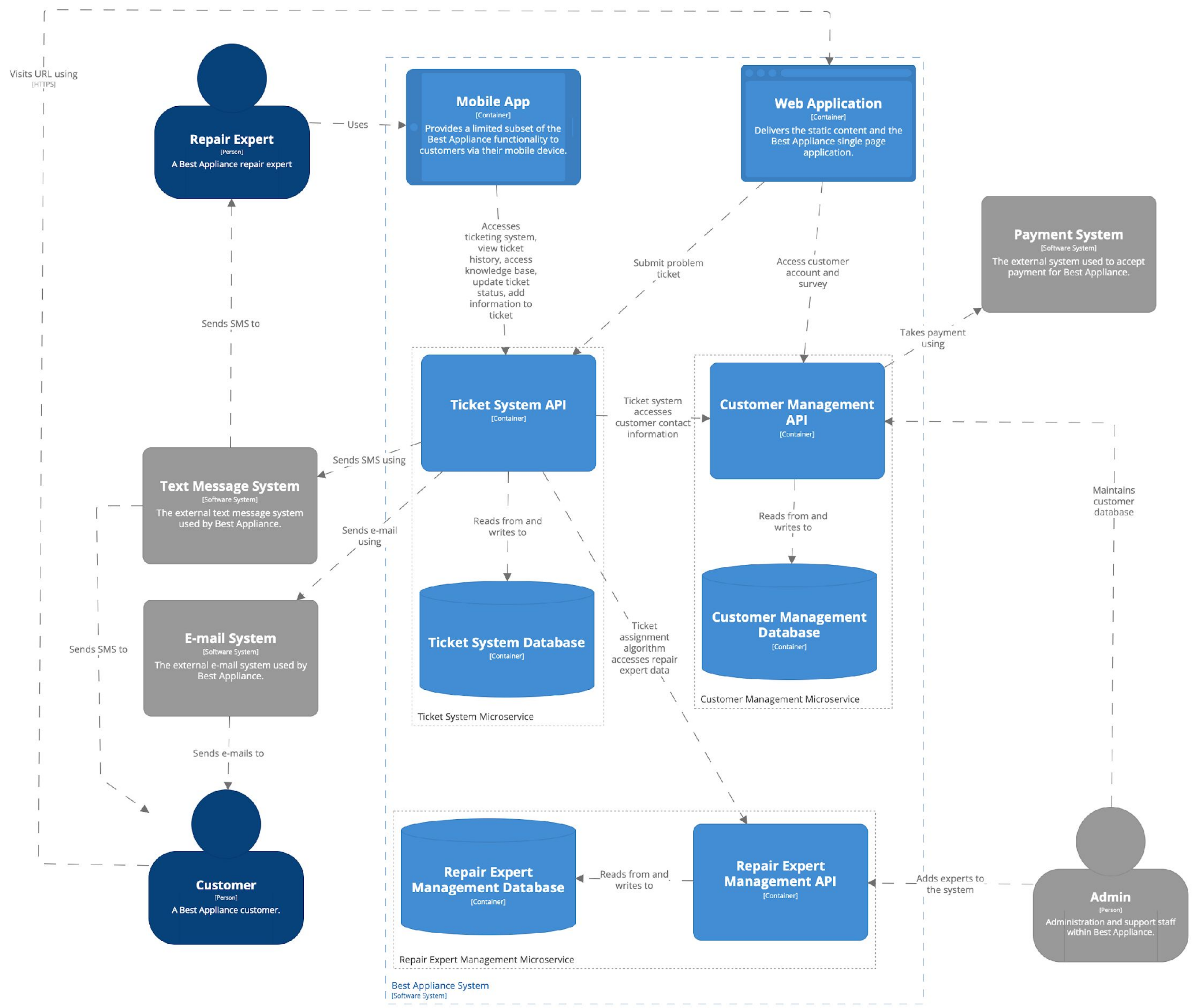
Architecture Diagram - Container

User interfaces:

- Mobile Application
- Web Application

Microservices:

- Ticket System
- Customer Management System
- Repair Expert System


Visits URL using
[HTTPS]
Repair Expert
[Person]
A Best Appliance repair expert
Uses
Mobile App
[Container]
Provides a limited subset of the Best Appliance functionality to customers via their mobile device.
Web Application
[Container]
Delivers the static content and the Best Appliance single page application.
Payment System
[Software System]
The external system used to accept payment for Best Appliance.
Accesses ticketing system, view ticket history, access knowledge base, update ticket status, add information to ticket
Submit problem ticket
Access customer account and survey
Takes payment using
Sends SMS to
Ticket System API
[Container]
Ticket system accesses customer contact information
Customer Management API
[Container]
Maintains customer database
Sends SMS using
Text Message System
[Software System]
The external text message system used by Best Appliance.
Sends e-mail using
Reads from and writes to
Reads from and writes to
E-mail System
[Software System]
The external e-mail system used by Best Appliance.
Sends SMS to
Ticket System Database
[Container]
Ticket assignment algorithm accesses repair expert data
Customer Management Database
[Container]
Customer Management Microservice
Ticket System Microservice
Sends e-mails to
Customer
[Person]
A Best Appliance customer.
Repair Expert Management Database
[Container]
Reads from and writes to
Repair Expert Management API
[Container]
Adds experts to the system
Admin
[Person]
Administration and support staff within Best Appliance.
Repair Expert Management Microservice
Best Appliance System
[Software System]

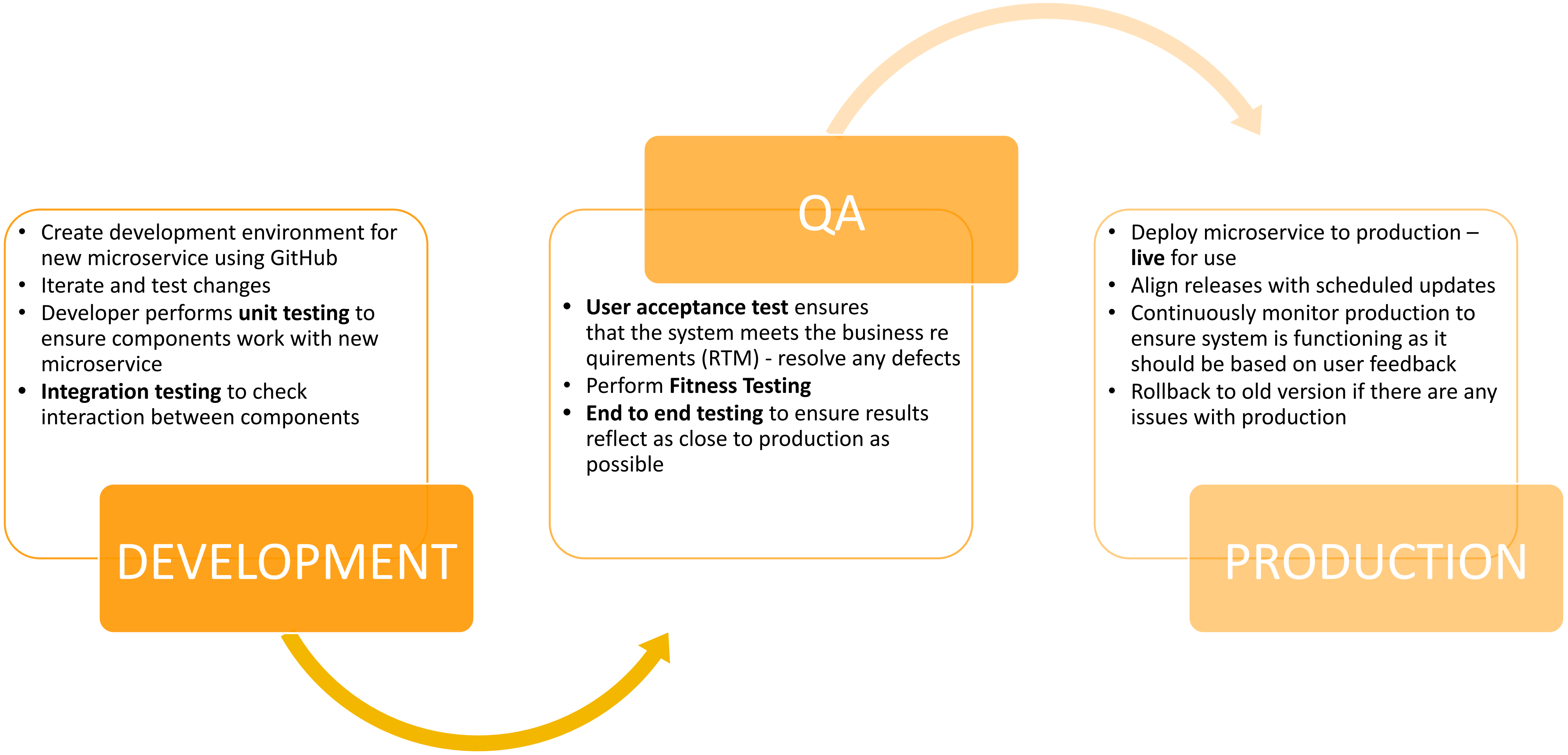
Rolling Deployment

With quality and budget being a priority, this deployment allows for incrementally deploying new versions of the application while balancing the risk and user disruption with new upgrades.

Decompose the monolith into loosely coupled components by breaking down the monolith into smaller more manageable microservices.

The new service is gradually and incrementally replacing the old version. This gradual approach helps maintain 24/7 accessibility to the service, as only a portion of the system is affected at any given time.

Rolling Deployment



DEVELOPMENT

- Create development environment for new microservice using GitHub
- Iterate and test changes
- Developer performs **unit testing** to ensure components work with new microservice
- **Integration testing** to check interaction between components

QA

- **User acceptance test** ensures that the system meets the business re quirements (RTM) - resolve any defects
- Perform **Fitness Testing**
- **End to end testing** to ensure results reflect as close to production as possible

PRODUCTION

- Deploy microservice to production – **live** for use
- Align releases with scheduled updates
- Continuously monitor production to ensure system is functioning as it should be based on user feedback
- Rollback to old version if there are any issues with production

Fitness Functions

To determine how fit a solution is and evaluate the performance to the desired problem the following is monitored during QA step in deployment:

Performance, Scalability, Fault Tolerance

- Monitor response time of notification to repair expert and email to customer.
- Monitor query results – do all historical records show up?

Performance, Elasticity, Reliability

- Monitor speed of web application and its availability during peak times.

Fault Tolerance, Scalability, Modularity

- Monitor if the ticket system algorithm assigns the correct expert to the problem ticket.

Scalability

- Monitor loose contracts - Use consumer-driven contracts to ensure loose contracts still contain enough information for the contract to function.

Thank You