UNIT – II

Software development models and DevOps: DevOps Lifecycle for Business Agility, DevOps, and Continuous Testing. **DevOps influence on Architecture:** Introducing software architecture, the monolithic scenario, Architecture rules of thumb, the separation of concerns, Handling database migrations, Microservices, and the data tier, DevOps, architecture, and resilience.

DevOps Lifecycle for Business Agility:

DevOps brings agility developments and release management through automation and continuous improvement.

The 7’c of DevOps is as follows.

1. Continuous business planning
2. Continuous Development
3. Continuous Integration
4. Continuous Deployment
5. Continuous Testing
6. Continuous Delivery and Monitoring
7. Continuous Feedback

1. Continuous business planning

Continuous planning is a business strategy that involves ongoing planning and delivery of new features, updates, and fixes. This approach enables businesses to prioritize their requirements and use them as inputs for their implementation teams. By utilizing agile processes and mature DevOps pipelines, organizations can continuously deliver these requirements. The key components of this process include effective requirement management tools, clear communication models, and an efficient feedback loop. This allows businesses to streamline their planning and implementation processes, delivering results quickly and efficiently.

2. Continuous Development

Continuous development is an umbrella to recognize end-to-end delivery of software covering continuous integration, continuous testing, continuous deployment, and continuous monitoring. This continuous development process is repeatable for all the features. Continuous development through a development pipeline ensures automation and streamlining of activities and processes required to deliver software.

Javascript, c/c++, Ruby and python are prominently used for coding applications in DevOps. The process of maintaining the code is called SCM(source code management), where version control tools such as GIT,TFS and Gitlab are used

3. Continuous Integration

Integration of unit steps in software coding stage is done through continuous integration. This stage is the heart of the entire DevOps life cycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. The source code gets modified several times and tose changes will happen weekly of daily bases. In this phase bugs in the source code are detected early on. Tools involved in CI(continuous integration): Jenkins, Apache Maven, Apache Tomcat, Sonarqube, Nexus /JFrog etc.

4.Continuous-Deployment

Deployment of developed software artifacts on different environments is automated through continuous deployment. It is also called as release automation. The new code is deployed continuously and configuration management tools play an essential role in executing tasks frequently and quickly. Some popular tools which are used in this phase, such as chef, puppet, Ansible and saltstack. Containerization tools are also playing an essential role in the deployment phase. Vagrant and Docker are popular tools used for this purpose.

5. Continuous Testing

Continuous testing (CT) is the process of executing automated test cases as part of the software delivery pipeline. Its goal is to obtain immediate and continuous feedback on the business risks associated with a software release candidate. Continuous feedback through continuous testing is one of the essential cores of Agile and DevOps process.

Key Benefits:

a. Quality at Speed: Continuous testing provides high-quality code at a faster speed. Release candidate code is always ready with a complete cycle of testing. Test automation avoids human errors.

b. Shift Left approach: Continuous testing promotes test earlier and more often in the delivery lifecycle. Testing should not be an activity toward the end of the life cycle. Unit test, system testing, user acceptance testing, service virtualization, regression testing and performance testing should all be performed at an early stage possibly as soon as the code is pushed to the development environment.

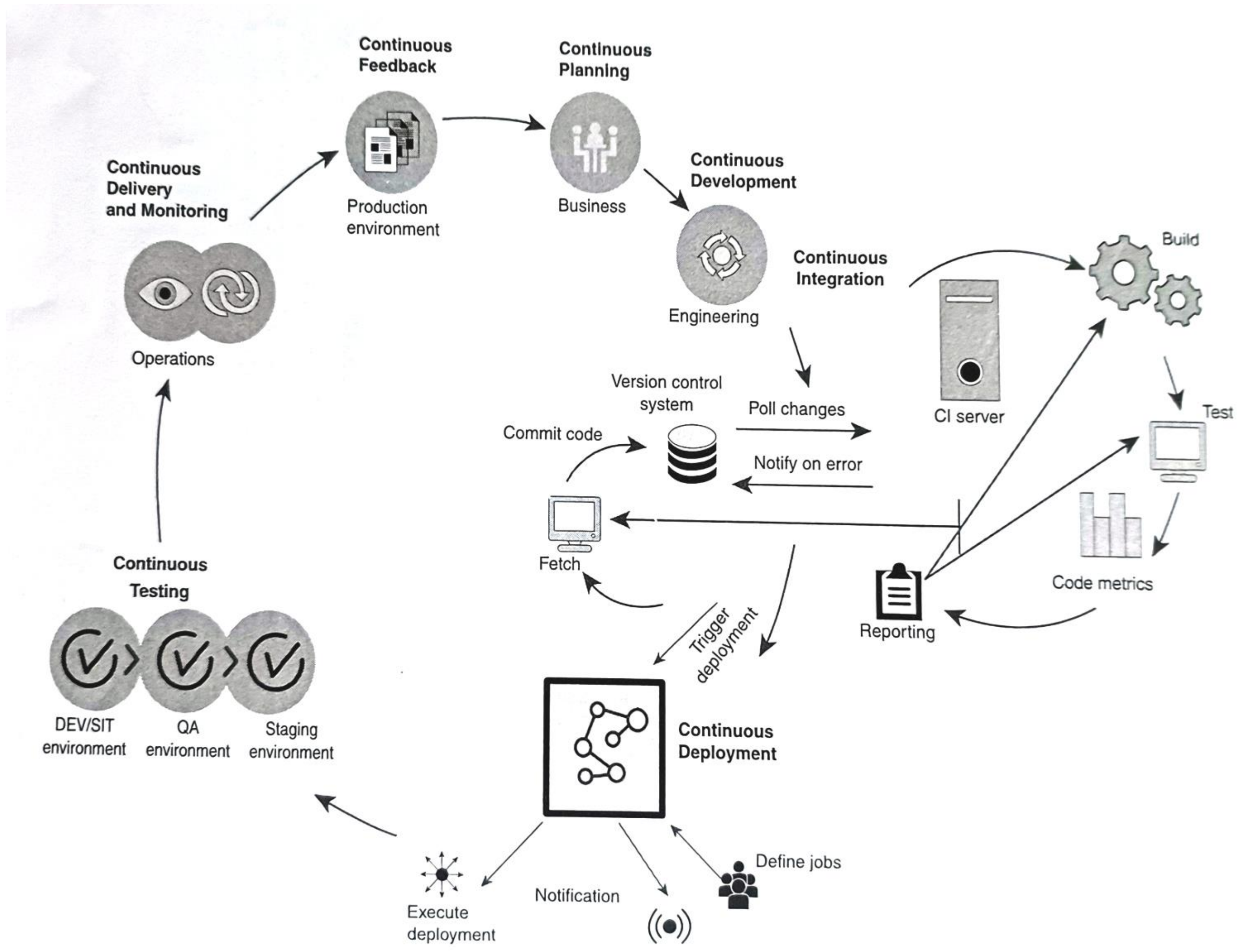
c. Faster Release: New software or change in software can quickly be delivered to the intended audience. It most suits to Agile and DevOps methodology where working software needs to deliver quickly.

d. Collaboration: Continuous testing and continuous feedback improve collaboration between development team, testing team, product owner and customer. Increase in collaboration improves product quality.

e. Cost Efficient: Continuous testing helps in early detection of defects and hence it has less impact on schedule and cost. Test automation reduces human efforts on a repetitive task. Test automation helps in better capacity planning and delivery speed.

f. Reduce Risk and Increase Confidence:

Continuous testing identifies the problem in the early phase of delivery - almost immediate of development. It not only validates the component/piece of code as an independent unit, but also its integration with rest of the application and other applications, that is, validation as a whole system.



DevOps Lifecycle Architecture for Business Agility

6 Continuous Delivery and Monitoring

Continuous delivery is delivery of any type of change - functional, configuration changes, defect fixes into production environments as soon as possible after completing all applicable processes. **Continuous monitoring** ensures application health is monitored continuously to keep a check on application's service to users and other applications, performance and security. Continuous monitoring focuses on production environment. Application, software, and infrastructure are monitored using different tools. It also involves analysis of monitoring data to identify trends, which can lead to application outages, and to take subsequent preventive and corrective actions. Monitoring also involves generation of different reports, which are input to analysis process. Dashboards and reports, generated by monitoring tools, are continuously reviewed by stakeholders to observe impact of changes and quality of service to end users.

7. Continuous Feedback

Continuous feedback is a process in which the feedback on multiple application aspects from different stakeholders and tools is given back to teams who can consider it in next iteration of release Application aspects include application performance, user experience, service level agreement (SLA), service quality, infrastructure performance, development process performance, etc. It can be gathered from different stakeholders such as end customers, operation teams, development teams, etc. How quickly feedback is captured and provided back to take corrective action, can define success of entire continuous feedback process. It also determines the cost organizations have to pay for the delayed

DevOps and Continuous Testing:

To achieve efficient and effective continuous testing is software development, it is essential to use automated testing tools in the DevOps pipeline. Manual testing can be limited and expensive. Automation is the only feasible solution in agile environment.

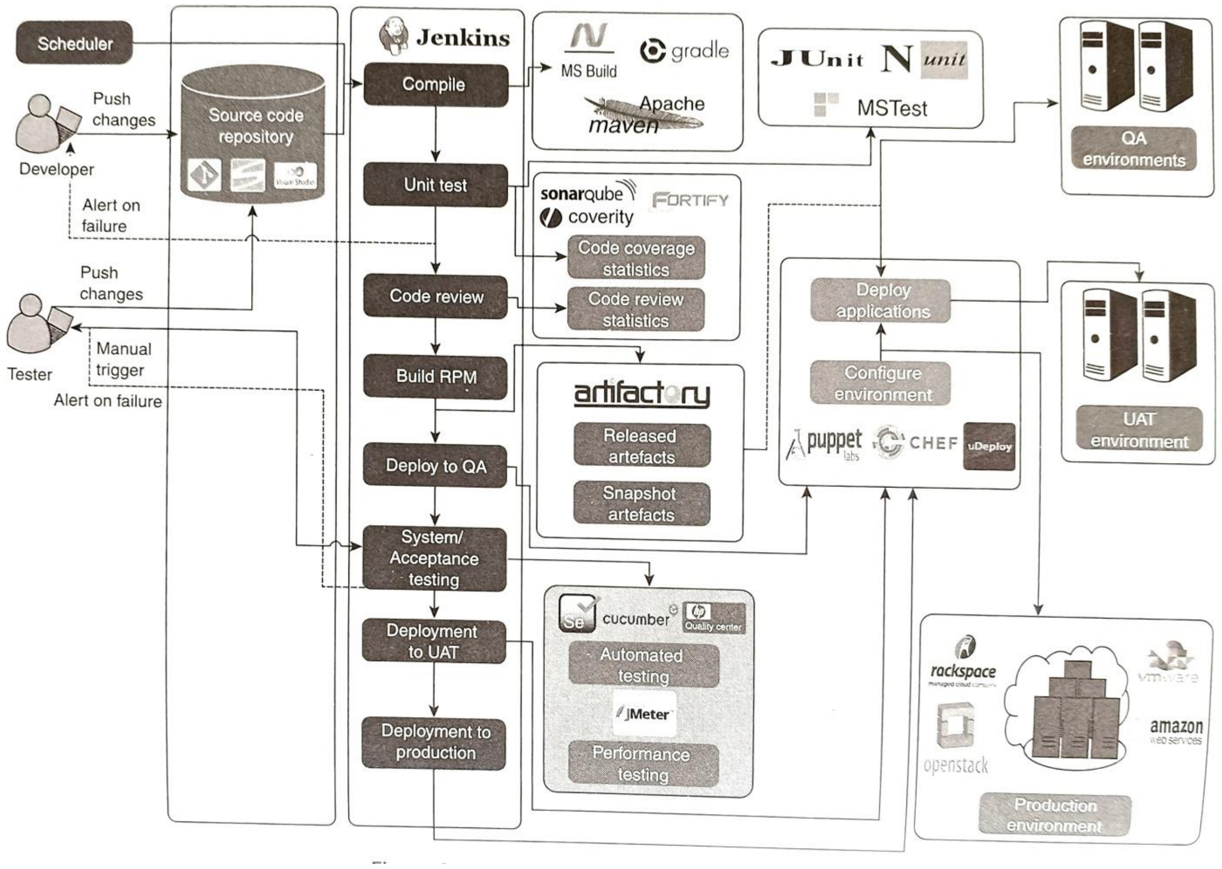
Following are the key points to consider:

1. Test automation and automated testing is a key aspect of DevOps.

2. Continuous testing promotes test early, test faster, test often and automate.

3. Continuous delivery needs continuous testing.

4. Continuous testing is the linchpin between continuos integeration and continuos delivery



The reference architecture in the above figure shows, how continuous testing is integrated into the CI-CD pipeline. The architecture includes various automated testing tools like selenium, cucumber, HP Quality center and JMeter for performance testing with HTTP requests.

DevOps influence on Architecture: Introducing software architecture, the monolithic scenario, Architecture rules of thumb, the separation of concerns, Handling database migrations, Microservices, and the data tier, DevOps, architecture, and resilience.

Introducing software architecture

We will discuss how DevOps affects the architecture of our applications rather than the architecture of software deployment systems. Often while discussing software architecture, we think of the non-functional requirements of our software. By non-functional requirements, we mean the different characteristics of the software rather than the requirements of particular behaviors.

A functional requirement could be that our system should be able to deal with credit card transactions. A non-functional requirement could be that the system should be able to manage several such credit cards transactions per second.

Here are two of the non-functional requirements that DevOps and CD place on software architecture:

1. We need to be able to deploy small changes often

2. We need to be able to have great confidence in the quality of our changes

The normal case should be that we are able to deploy small changes all the way from developers' machines to production in a small amount of time. Rolling back a change because of unexpected problems caused by it should be a rare occurrence.

So, if we take out the deployment systems from the equation for a while, how will the architecture of the software systems we deploy be affected?

The monolithic scenario

If all the functionalities (frontend, backend and configuration files) of a project exist in a single code base, then that application is known as monolithic application.

One way to understand the issues that a problematic architecture can cause for CD is to consider a counter-example for a while.

Let's suppose we have a large web application with many different functions. We also have a static website inside the application. The entire web application is deployed as a single Java EE application archive. So, when we need to fix a spelling mistake in the static website, we need to rebuild the entire web application archive and deploy it again.

While this may be seen as a silly example, and the enlightened reader would never do such a thing, I have seen this anti-pattern live in the real world. As DevOps engineers, this could be an actual situation that we may be asked to solve.

Let's break down the consequences of this tangling of concerns. What happens when we want to correct a spelling mistake? Let's take a look:

1. We know which spelling mistake we want to correct, but in which code base do we need to do it? Since we have a monolith, we need to make a branch in our code base's revision control system. This new branch corresponds to the code that we have running in production.

2. Make the branch and correct the spelling mistake.

3. Build a new artifact with the correction. Give it a new version.

4. Deploy the new artifact to production.

Ok, this doesn't seem altogether too bad at first glance. But consider the following:

We made a change to the monolith that our entire business critical system comprises. If something breaks while we are deploying the new version, we lose revenue by the minute. Are we really sure that the change affects nothing else?

It turns out that we didn't really just limit the change to correcting a spelling mistake. We also changed a number of version strings when we produced the new artifact. But changing a version string should be safe too, right? Are we sure?"

The point here is that we have already spent considerable mental energy in making sure that the change is really safe. The system is so complex that it becomes difficult to think about the effects of changes, even though they may be trivial.

Now, a change is usually more complex than a simple spelling correction. Thus, we need to exercise all aspects of the deployment chain, including manual verification, for all changes to a monolith. We are now in a place that we would rather not be.

Disadvantages:

1. Continuous deployment is difficult

2. You must redeploy the entire application on each update.

3. It is difficult to modify or update the application

4. It requires a large development team to work together on a single code base

5. In monolithic all components are tightly coupled together which makes it difficult to isolate and scale individual components.

Architecture rules of thumb

There are several architecture principles that can help teams design software systems that are well-suited to DevOps and continuous delivery. Here are some examples:

1. **Microservices architecture**: Microservices architecture is an approach to software architecture where a single application is broken down into a collection of small, independently deployable services. This approach allows for greater flexibility and scalability, as each service can be developed, tested, and deployed independently.

2. **Service-oriented architecture** (SOA): SOA is an architectural style that uses a set of loosely coupled services to accomplish specific tasks. These services communicate with each other through well-defined interfaces and can be developed, tested, and deployed independently. This approach allows for greater agility and flexibility in software development.

3. **Continuous integration and delivery (CI/CD):** CI/CD is a set of practices that involves continuously building, testing, and deploying software changes. This approach requires a well-architected system that can handle frequent changes and allows for easy testing and deployment.

4. **Infrastructure as code**: Infrastructure as code (IaC) is an approach to infrastructure management where infrastructure resources are defined in code and versioned alongside the application code. This approach allows for greater consistency and repeatability in infrastructure management, which is essential for DevOps and CD.

5. **Containerization**: Containerization is a method of packaging software so that it can be run consistently across different environments. This

approach allows for greater portability and flexibility in software deployment.

Overall, these architecture principles can help teams design software systems that are well-suited to the needs of DevOps and continuous delivery. By following these principles, teams can build systems that are flexible, scalable, and resilient, and that can handle frequent changes with confidence.

The separation of concerns

The separation of concerns is a fundamental principle in software design that was first introduced by Dutch computer scientist Edsger Dijkstra in 1974. It states that we should consider different aspects of a system separately in order to organize our thoughts efficiently. This idea is arguably the most important rule in software design and has led to the development of many other well-known rules in the field. By separating concerns, we can simplify the design process and create more modular, flexible, and maintainable software systems.

Handling database migrations

Handling changes in a relational database requires special consideration.

A relational database stores both data and the structure of the data. Upgrading a database schema offers other challenges than the ones present when upgrading program binaries. Usually, when we upgrade an application binary, we stop the application, upgrade it, and then start it again. We don't really bother about the application state. That is handled outside of the application.

When upgrading databases, we do need to consider state, because a database contains comparatively little logic and structure, but contains much state.

To describe a database structure change, we issue a command to change the structure.

The database structures before and after a change is applied should be seen as different versions of the database. How do we keep track of database versions?

Liquibase is a database migration system that, at its core, uses a tried and tested method. There are many systems like this, usually at least one for every programming language. Liquibase is well-known in the Java world, and even in the Java world, there are several alternatives that work in a similar manner. Flyway is another example for the Java platform.

Generally, database migration systems employ some variant of the following method:

1.Add a table to the database where a database version is stored.

2.Keep track of database change commands and bunch them together in versioned change sets. In the case of Liquibase, these changes are stored in XML files. Flyway employs a slightly different method where the change sets are handled as separate SQL files or occasionally as separate Java classes for more complex transitions.

3.When Liquibase is being asked to upgrade a database, it looks at the metadata table and determines which change sets to run to make the database up-to-date with the latest version.

As previously stated, many database version-management systems work like this. They differ mostly in the way the change sets are stored and how they determine which change sets to run. They may be stored in an XML file, as in the case of Liquibase (Liquibase supports several formats, such as XML, YAML, JSON, and SQL), or as a set of separate SQL files, as with Flyway. This later method is more common with homegrown systems and has some advantages. The Clojure ecosystem also has at least one similar database migration system of its own, called Migratus.

The following table highlights a handful of example schema migration tools:

Language	Name	Site
Java	Liquibase	http://www.liquibase.org/
Java	Flyway	https://flywaydb.org/
Clojure	Migratus	https://github.com/yogthos/migratus
Python	Alembic	https://pypi.python.org/pypi/alembic

Three-tier systems

In this architecture, the entire application is organized into three computing tiers

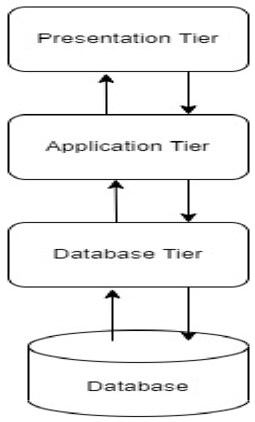
- Presentation tier
- Application tier
- Data-tier

The major benefit of the three tiers in client-server architecture is that these tiers are developed and maintained independently and this would not impact the other tiers in case of any modification. It allows for better performance and even more scalability in architecture can be made as with the increasing demand, more servers can be added.

- **Presentation Tier:** It is the user interface and topmost tier in the architecture. Its purpose is to take request from the client and displays information to the client. It communicates with other tiers using a web browser as it gives output on the browser. If we talk about Web-based tiers then these are developed using languages like- HTML, CSS, JavaScript.

- **Application Tier:** It is the middle tier of the architecture also known as the logic tier as the information/request gathered through the presentation tier is processed in detail here. It also interacts with the server that stores the data. It processes the client's request, formats, it and sends it back to the client. It is developed using languages like-Python, Java, PHP, etc.

- **Data Tier:** It is the last tier of the architecture also known as the Database Tier. It is used to store the processed information so that it can be retrieved later on when required. It consists of Database Servers like-Oracle, MySQL, DB2, etc. The communication between the Presentation Tier and Data-Tier is done using middle-tier i.e. Application Tier.



Advantages:

- Developers are independent to update the technology of one tier as it would not impact the other tiers.

- Reliability is improved with the independence of the tiers as issues of one tier would not affect the other ones.

- Programmers can easily maintain the database, presentation code, and business/application logic separately. If any change is required in business/application logic then it does not impact the presentation code and codebase.

- Security is improved as the client cannot communicate directly with Database Tier.
- The integrity of data is maintained.

Disadvantages:

- The Presentation Tier cannot communicate directly with Database Tier.
- Complexity also increases with the increase in tiers in architecture.
- There is an increase in the number of resources as codebase, presentation code, and application code need to be maintained separately.

Microservices

Microservice architecture, also known as ‘microservices,’ is a development method that breaks down software into modules with specialized functions and detailed interfaces.
Microservices allow large applications to be split into smaller pieces that operate independently.

Characteristics Of Microservices

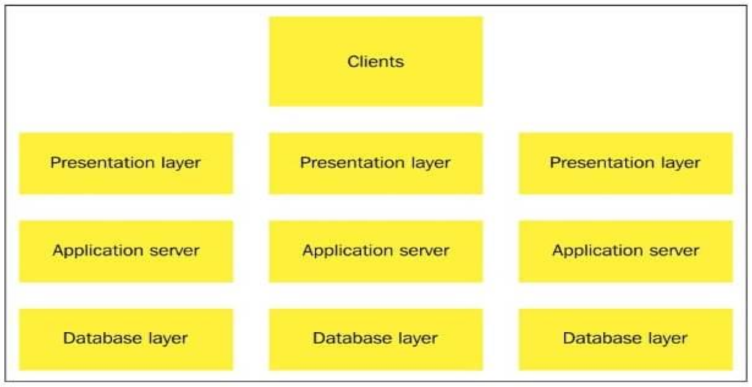
- The application will be divided into micro-components
- Each service has a separate database layer, independent codebase, and CI/CD tooling sets
- Microservices can be written in a variety of programming languages, and frameworks, and each service act as a mini-application on its own.
- It contains APIs
- It performs Decentralized operations
- It is Designed for business applications
- Each Service can implement an independent security mechanism

Advantages:
The application is broken into smaller modules that are easy for developers to code and maintain.
It is easy to scale
It Removes dependency
Each service has its own database.
Provides the security for application.

Microservices is a recent term used to describe systems where the logic tier of the three-tier pattern is composed of several smaller services that communicate with language-agnostic protocols.

Typically, the language-agnostic protocol is HTTP based, commonly JSON REST, but this is not mandatory. There are several possibilities for the protocol layer.

This architectural design pattern lends itself well to a CD approach since, as we have seen, it's easier to deploy a set of smaller standalone services than a monolith.
Here is an illustration of what a microservices deployment may look like:



Microservices and the data tier

One way of viewing microservices is that each microservice is potentially a separate three-tier system. We don't normally implement each tier for each microservice, though. With this in mind, we see that each microservice can implement its own data layer. The benefit would be a potential increase of separation between services.

It is more common in my experience, though, to put all of the organization's data into a single database or at least a single database type. This is more common, but not necessarily better.

There are pros and cons to both scenarios. It is easier to deploy changes when the systems are clearly separate from each other. On the other hand, data modeling is easier when everything is stored in the same database.

DevOps, architecture, and resilience

We have seen that the microservice architecture has many desirable properties from a DevOps point of view. An important goal of DevOps is to place new features in the hands of our user faster. This is a consequence of the greater amount of modularization that microservices provide.

Those who fear that microservices could make life uninteresting by offering a perfect solution without drawbacks can take a sigh of relief, though. Microservices do offer challenges of their own.

We want to be able to deploy new code quickly, but we also want our software to be reliable.
Microservices have more integration points between systems and suffer from a higher possibility of failure than monolithic systems.

Automated testing is very important with DevOps so that the changes we deploy are of good quality and can be relied upon. This is, however, not a solution to the problem of services that suddenly stop working for other reasons. Since we have more running services with the microservice pattern, it is statistically more likely for a service to fail.

We can partially mitigate this problem by making an effort to monitor the services and take appropriate action when something fails. This should preferably be automated.
In our customer database example, we can employ the following strategy:

1.We use two application servers that both run our application
2.The application offers a special monitoring interface via JsonRest
3.A monitoring daemon periodically polls this monitoring interface
4.If a server stops working, the load balancer is reconfigured such that the offending server is taken out of the server pool

This is obviously a simple example, but it serves to illustrate the challenges we face when designing resilient systems that comprise many moving parts and how they may affect architectural decisions.

Why do we offer our own application-specific monitoring interface, though? Since the purpose of monitoring is to give us a thorough understanding of the current health of the system, we normally monitor many aspects of the application stack. We monitor that the server CPU isn't overloaded, that there is sufficient disk and memory space available, and that the base application server is running. This may not be sufficient to determine whether a service is running properly, though. For instance, the service may for some reason have a broken database access configuration. A service-specific health probing interface would attempt to contact the database and return the status of the connection in the return structure.

It is, of course, best if your organization can agree on a common format for the probing return structure. The structure will also depend on the type of monitoring software used.