# Глава 1. Проблеми на информационното осигуряване при управление на поръчките от клиенти

## Управление на веригите от поръчки и доставки и тяхното приложение в системите за планиране на ресурси

В литературата съществуват множество различни дефиниции за термина „верига на доставките“. Според Chopra and Meindl „веригата за доставки се състои от всички етапи, които пряко или непряко участват в изпълнението на заявките на клиента. Веригата на доставки включва не само производителя и доставчиците, но и превозвачите, складовете, търговците на дребно и самите клиенти“. Ganeshan and Harrison, пък дефинират веригата за доставки, като: „мрежа от съоръжения и възможности за дистрибуция, която изпълнява функциите на доставка на материали, превръщането на тези материали в междинни и готови продукти и разпространението на тези готови продукти на клиентите.“. Друга дефиниция, която откриваме „веригата за доставки е съвкупност от процеси и ресурси, необходими за извършване и доставка на продукт на крайния потребител“ или също „канал за ефективно движение на материали, продукти, услуги или информация от доставчици към клиенти“. Настоящия труд се спира на определението на Милушева, П., която дефинира понятието, като „ясно очертана верига от свързани двойки логистични звена „доставчик – получател“ (структурирани подразделения на фирмата и/или логистичните й партньори), по която конкретната стока и/или услуга се доставя на крайния потребител в съответствие с неговата заявка и изисквания“.

Когато говорим за движението на стоките през веригата за доставки, нормалното развитие е от производителя към крайния потребител. Обратната логистика, от друга страна, представлява обратното движение на стоките - от крайния потребител към производителя или друга инстанция. Обратната логистика включва различни дейности, като връщане на стоки от страна на потребителите, които по някаква причина не отговарят на техните очаквания, връщане на стоки с цел ремонт, рециклиране, замяна и други (Plaza-Úbeda et al., 2020). Какво точно ще се случи с тези върнати стоки - дали ще бъдат подлагани на ремонт и след това препродажба или ще бъдат рециклирани, зависи както от характеристиките на самите продукти, така и от причините за тяхното връщане. В книгата си "Обратни вериги на доставка," Surendra Gupta сравнява основните сходства и разлики между нормалната и обратната верига за доставки, като подчертава основните аспекти, които ги различават. Тези аспекти са описани и сравнени в подробности в съответната таблица в книгата.

Таблица 1.1.  
 Сравнение между права и обратна верига за доставки   
(разработка на автора)

|  |  |
| --- | --- |
| **Права верига за доставки** | **Обратна верига за доставки** |
| Базирана на оптимизиране на печалбата и разходите | Базирана на екологичните принципи и закони, както и на оптимизирането на печалбите и разходите |
| Сравнително по-лесно и ясно прогнозиране на търсенето на продукти | По-трудно прогнозиране за връщане на продукти |
| По-малко вариации в качеството на продукта | Високи различия |
| Времето и стъпките за обработка са добре дефинирани | Времето и стъпките за обработка зависят от състоянието от върнатия продукт |
| Стоките се транспортират от едно място до много други места | Върнатите продукти се събират от много места и пристигат в едно преработвателно предприятие |
| Оценката на разходите е по-лесна заради счетоводните системи | Определянето и представянето на разходите е сложно |
| Скоростта е конкурентно предимство | Скоростта не е критичен фактор |
| Опаковката на продуктите е стандарта | Опаковката е високо променлива |
| Стандартна структура на самият продукт | Структурата на продукта е модифицирана |
| Прозрачност на процесите, дължащи се на проследяването на продуктите в реално време | По-малко видими процеси, поради липсата на възможности за информационна |

Може да обобщим, че правите вериги за доставки се фокусират върху печалбата и оптимизирането на разходите, докато обратните вериги за доставки дават приоритет на спазването на законите и обратната връзка от клиента. Те се различават по прогнозиране на търсенето, вариации в качеството на продукта, времена за обработка, транспортиране на стоки, оценка на разходите, конкурентно предимство, опаковка на продукта, структура на продукта и прозрачност на процеса.

Важен елемент в управлението на веригата за доставки е логистиката. European Logistics Association (ELA) дефинира логистиката като "организация, планиране, контрол и реализация на придвижването на стоковия поток от проектирането и закупуването, през производството и разпределението до крайния потребител с цел удовлетворяване изискванията на пазара с минимални операционни и капиталови разходи". Според Павлов, П. логистиката се определя като процес, който включва планиране, реализация и контрол на движението и съхранението на материалите и свързаната с тях информация. Тези дейности се идентифицират по различни параметри като размерност, начална и крайна точка на движение, дължина на пътуването, скорост, време на движението, време на престой, вид на използваните транспортни средства, условия на транспортиране. В тази смисъл, Schachenhofer (2023) разглежда логистичната система като устойчива мрежа от звена, които са взаимно свързани и управлявани централно чрез административни системи, които подпомагат управлението на целия логистичен процес. Целта е да се удовлетворят заявките и нуждите на клиентите, като се поддържа баланс между предлагането и търсенето.

В тази връзка се среща понятието „информационен поток“, което представлява обмен на данни, които се изпращат устно, чрез документи или по друг начин и са свързани с материалния поток. Този информационен обмен играе роля в логистичната система и е от важно значение за управлението на веригата за доставки. Един от най-често срещаните информационни потоци в логистиката е потокът на "поръчки от клиенти" (Zając & Świeboda, 2023). Всеки бизнес организира този процес по индивидуален начин, съобразено със специфичните си нужди и процедури.

В изследователската поредица Journal of Business Logistics се свързват стратегията и планирането на веригата за доставки и конкурентоспособността на производствени предприятия на световния пазар за строителни материали. Авторите предоставят информация, която помага да се определи кога са необходими по-големи запаси или кои продукти трябва да бъдат предлагани в определен момент, чрез анализ на предлагането и търсенето. Интегрирането на планирането и изпълнението на процесите в рамките на веригата за доставки включва планиране на търсенето, управление на доставките, производство, контрол на запасите, складиране, транспортиране и други логистични операции (Alzoubi et al., 2020). Този интегриран подход помага за оптимизиране на потока от материали, информация и финансови ресурси, както и за управление на връщането на излишни или дефектни продукти. Според Văcar (2019) логистичният мениджмънт включва планиране, организиране, координация и контрол на всички операции, които да удовлетворят изискванията на клиентите, като осигурят ефективно движение на стоките от точката на зареждане до точката на доставка. Логистичният мениджмънт включва вземане на стратегически, тактически и оперативни решения, свързани с развитието на логистичната дейност и взаимодействието с доставчиците и другите участници в веригата за доставки. Bardakci (2020) допълва, че стратегическият логистичен план има за цел да реализира поставената стратегия и да осигури ефективното функциониране на логистичната мрежа. Според някои изследвания, оптимизационните задачи в логистичното планиране могат да се разграничат на различни функционални области, като една от тях е управлението на поръчките. Този аспект включва регламентиране и оптимизация на всички етапи в цикъла на изпълнение на поръчките, включително приемането, обработката и доставката им (Calabrò et al., 2020). В този контекст, изборът на подходящи технически средства и технологии за приемане, обработка, въвеждане на електронен обмен на данни и установяването на параметри за качество на обслужване също са аспекти на управлението на поръчките. Ефективното управление на материалните и съпътстващите ги потоци изискват координирано изпълнение на разнообразни функции и операции в рамките на логистичната система. Тази координация се изпълнява както на стратегическо, така и на оперативно ниво, тъй като влияе както на ритмичността на бизнес дейността, така и на ефективността на самата логистика.

В този смисъл, производството е компонента на веригата за доставки, който представлява процедура на трансформация на основни материали в готови стоки. Тази процедура е насочена към създаване на стойност както за производителите, така и за потребителите. Съществуват различни стратегии за управление на производството, като проектиране по поръчка, производство по поръчка, сглобяване по поръчка и производство на склад. Всяка от тези стратегии има свои предимства и недостатъци, но целта им е обща - да създадат стойност както за клиентите, така и за самата организация. С прилагането на подходящата стратегия, предприятията могат да подобрят качеството и доставката на своите продукти, увеличат ефективността и точността, като същевременно се намаляват излишните запаси и разходите, което в крайна сметка е от полза както за клиентите, така и за бизнеса. Планирането на производството съгласува търсенето с производствения капацитет и определя график за доставка на готови продукти (Chen, 2020).

Възможните компоненти, които влияят върху SCM, могат да бъдат обобщени в следните групи: управление на жизнения цикъл на продукта (PLCM), логистика, стратегия за веригата за доставки, планиране на веригата за доставки (SCP), доставки и управление на активи. Те са представени графично на фиг. 1.1.



Фиг 1.1. Модел на елементите, съставящи управлението на веригите за доставки. Източник: Meehan, J., & Pinnington, B., 2021, Адаптирано от автора

Този модел представлява обобщение на информацията, свързана с движението на материали и финансовите процеси от момента на производство до достигането на продуктите и услугите до крайните клиенти, включвайки свързани мрежи, канали и предприятия, които си сътрудничат. Статия, представена от Journal of Supply Chain Management описва няколко фундаментални аспекта на този модел:

- Увеличаване на печалбата: Оптимизирането на процесите във веригата за доставки може да доведе до по-голяма ефективност и по-ниски оперативни разходи, което накрая води до увеличаване на печалбата (Goodman, 2019);

- Увеличаване на паричния поток: Правилното управление на веригата за доставки може да подобри паричния поток на организацията, като ускори оборота на стоките и оптимизира управлението на финансовите ресурси;

- Подобряване на обслужването на клиентите: Операциите в рамките на веригата за доставки са насочени към предоставянето на добро обслужване на клиентите, като осигурява, продуктите да се доставят навреме и да отговарят на изискванията;

- Намаляване на оперативните разходи: Ефективното управление на веригата за доставки може да намали оперативните разходи, свързани с транспорта, складирането и обработката;

Като обобщение на посочените по-горе фактори SCM се свързва с поддържането на непрекъснати доставки, управлението на договорните задължения, запазването на връзките с клиенти, както и поддържането на конкурентно предимство на пазара. Според Katsaliaki et al., 2021 чрез внедряването на SCM системи, производствените предприятия могат да подобрят производителността си. Модулите на тези системи се интегрират взаимно за да допринесат за ефективното управление на модела, представен по горе. В крайна сметка контролът на разходите и качественото обслужване на клиентите, подобрява конкурентоспособността и устойчивост на пазара.

На базата на някои от последните класации за компании за строителни материали към 1 декември 2023 г., внедряването на стабилни системи за SCM е от първостепенно значение за поддържане и интеграция на бизнес процесите от край до край. CRH plc, Vulcan Materials Company, Martin Marietta Materials, Inc., Anhui Conch Cement и Heidelberg Materials AG използват водещ в индустрията SCM софтуер като SAP S/4HANA, Oracle SCM Cloud, Blue Yonder, Microsoft Dynamics 365, Kinaxis RapidResponse като същевременно използват и персонализирани решения, съобразени с техните логистични и оперативни проблеми. Тези системи са проектирани да оптимизират доставките, да рационализират производствените графици, да осигурят ефективни дистрибуторски мрежи, да подобряват вземането на решения, да насърчават сътрудничеството на доставчици, диспечери и клиенти и стимулират инициативи за устойчивост.

## 1.2. Възможности за централизация и .. на процесите по управление чрез прилагане на облачни технологии

През последните години облачните технологии се превърнаха във водеща тенденция в софтуерната индустрия. Текущите изследвания на водещи специалисти в областта определят изчисленията в облак като съвременен начин за изграждане на корпоративни системи, използвайки съвременните практики за разработка на високо-качествен софтуер и инфраструктура. Факторите като проектиране, интегриране и внедряване на система пряко влияят на процеса по управление на поръчките от клиенти. От технологична гледана точка облачно базираните решения са проектирани да приемат бързо промените, да обслужват голям набор от хора и да бъдат устойчиви на всякакъв вид натоварване или хакерски атаки (Vettor, 2022). Oбщото разбиране на облачната концепцията се различава, като например организацията Cloud Native Computing Foundation (CNCF) предлага следната дефиниция: "*Технологиите, базирани на облак, дават възможност на организациите да създават и изпълняват приложения в модерни, динамични среди като публични, частни и хибридни облаци, чрез мрежи от услуги и микроуслуги. Качества на системите са устойчивост, висока наличност и достъпност, мащабируемост и управляемост, които са от критично значение за много от бизнес единиците. Автоматизацията на тези процеси позволява на инженерите да правят промени, с голямо въздействие, но с минимални усилия."* От друга страна National Institute of Standards and Technology (NIST) определя облачните изчисления като „*модел за позволяване на мрежов достъп, при поискване, до споделен пул от конфигурируеми изчислителни ресурси, които могат бързо да бъдат предоставени и внедрени с минимални усилия.“* Разгледаните определения дават различни тълкувания, като общо схващане е, че облачно базираните системи се свързват главно с бързина (Smith, 2022). Освен да задоволят своите клиенти, производствените предприятия трябва и да се съревновават с конкурентите, което се свързва с темата за незабавното и безпроблемно пускане на иновативни идеи на пазара (Тодоранова, 2016). В тази връзка, облачните технологии позволяват на компаниите да реагират на пазарните условия като постоянно актуализират малки, но важни области.

Текущите изследвания на водещи специалисти анализират редица фактори, които благоприятстват скоростта на облачните носители. Методологията на дванадесетте фактора (The twelve-factor), представена в таблица 1.3, описва набор от принципи и практики, които разработчиците се очаква да следват, за да създадат приложения, оптимизирани за модерни облачни среди. Много практици смятат Twelve-Factor за солидна основа за изграждане на облачни системи, защото е приложима за всяко уеб, десктоп или мобилно базирано решение. Системите, изградени по тези принципи, могат да се внедряват и мащабират, като същевременно позволяват добавяне на нови или да промяна на съществуващи функционалности.

Таблица 1.3.  
 Обобщение на методологията на дванадесетте фактора  
(адаптирано от автора)

| **Фактор** | **Обяснение** |
| --- | --- |
| **Code Base** | Единична база за сорс кода на всяка микроуслуга, съхранявана в собствено хранилище към GitHub, GitLab, Azure DevOps и други. Проследен с контрол на версиите, всяка микроуслуга може да се внедри в множество среди (QA, Staging, Production). |
| **Dependencies** | Всяка микроуслуга изолира и пакетира свои собствени зависимости, като обхваща промени, които да не засягат цялата система. |
| **Configurations** | Конфигурационната информация се управлява чрез инструмент, извън кода на микроуслугата. Тя може да бъде различна за различните страни. |
| **Backing Services** | Допълнителните ресурси (хранилища за данни, кешове, брокери на съобщения) трябва да бъдат изложени чрез адресируем URL адрес. Това отделя ресурса от приложението, което му позволява да бъде взаимозаменяем. |
| **Build, Release, Run** | Всяка нова версия трябва да мине през през няколко етапа на изграждане, внедяване и изпълнение. Всеки от тях трябва да бъде маркиран с уникален идентификационен номер и да поддържа възможност за обратно връщана. Съвременните CI/CD системи спомагат за изпълнението на този принцип. |
| **Processes** | Всяка микроуслуга трябва да се изпълнява в свой собствен процес, изолиран от другите услуги. |
| **Port Binding** | Всяка микроуслуга трябва да бъде самостоятелна със своите интерфейси и функционалност, изложени на собствен порт. Това осигурява изолация от други микросервизи, но и позволява пренасочване на портове. |
| **Concurrency** | Когато капацитетът на микроуслуга трябва да се увеличи, мащабирането се случнва хоризонтално, ориентирано към множество идентични процеси (копия). |
| **Disposability** | Екземплярите на услугите трябва да благоприятстват бързото стартиране, както и изключване. Контейнерите заедно с оркестратор, по своята същност, отговарят на това изискване. |
| **Dev/Prod Parity** | Различните среди е необходимо да се поддържат възможно най-сходни, през целия жизнен цикъл на приложението. Тук контейнеризацията може значително да допринесе чрез насърчаването на същата среда за изпълнение. |
| **Logging** | Регистрационните файлове, генерирани от микросервизи, трябва да се третират като потоци от събития, обработвани с агрегатор. Инструменти за управление на логове (като Azure Monitor или Splunk) се препоръчват за разпространяването на лог данни и тяхното дългосрочно архивиране. |
| **Admin Processes** | Изпълняване на административни задачи, като почистване на вътрешни данни или рестартиране на услуга. |

В книгата „Beyond the Twelve-Factor App „ авторът Кевин Хофман описва подробно всеки от оригиналните 12 фактора, като добавя три допълнителни, които отразяват модерен дизайн на облачни приложения.

Таблица 1.4.  
 Допълнение на методологията на дванадесетте фактора  
(адаптирано от автора по Hoffman, 2016)

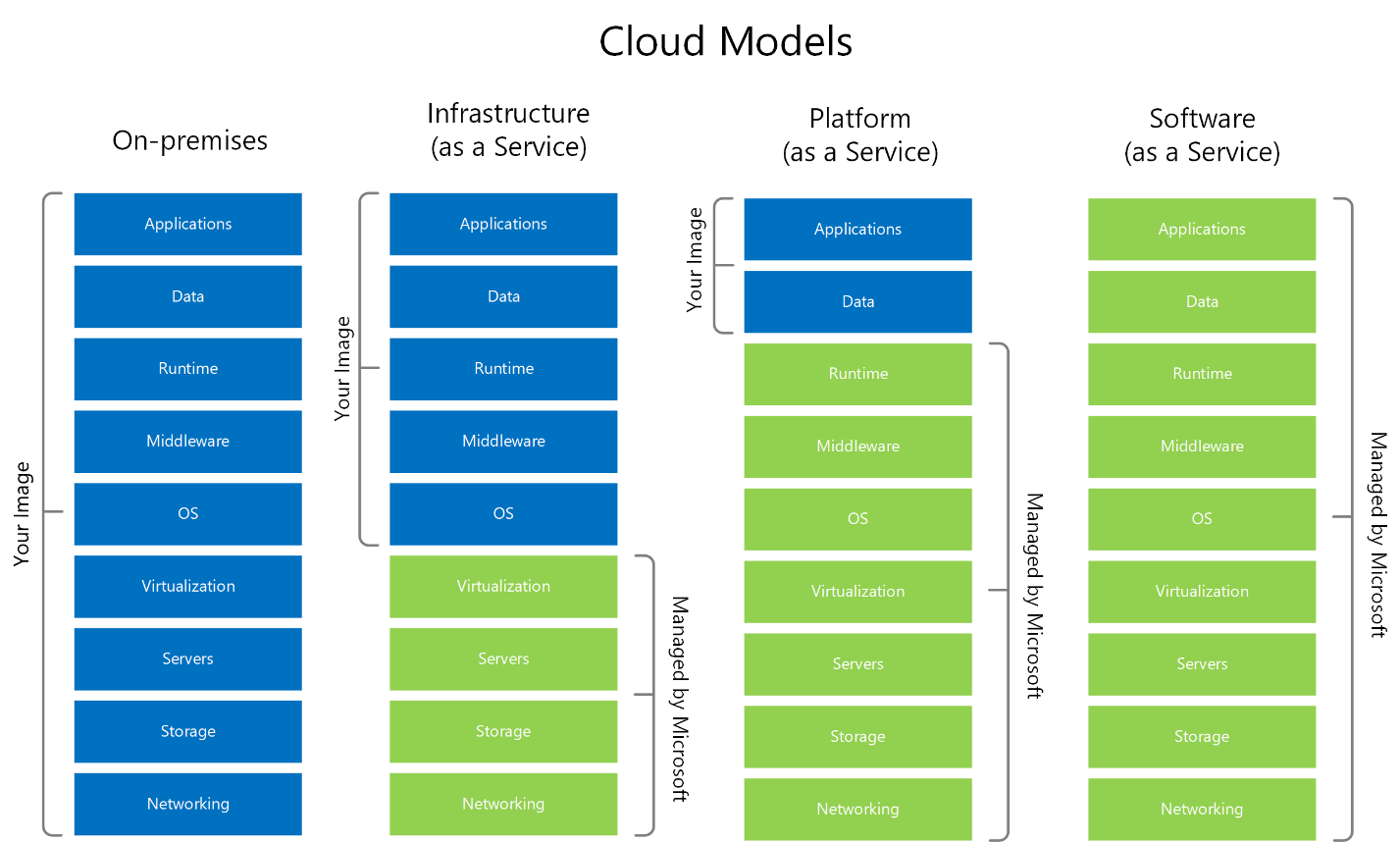
| **Фактор** | **Обяснение** |
| --- | --- |
| **API First** | Всеки ресурс трябва да бъде разгледан като приложно-програмен интерфейс, който да бъде интегриран към основната система. |
| **Telemetry** | Дизайнът на системата трябва да включва събирането на специфични за домейна данни, както и за състоянието на системата. |
| **Authentication/ Authorization** | Прилагане на идентичност и базиран контрол на достъпа. |

В този смисъл, Microsoft Well-Architected Framework (Stanford D. et al, 2022) предоставя набор от ръководни принципи, които се използват за подобряване качеството на работното натоварване. Следната таблица представя пет стълба на т.н. „добра архитектурата“:

Таблица 1.4.  
Стандартни за добри практики на облачната индустрия  
(адаптирано от автора по АБВ, 2022)

| **Принципи** | **Описание** |
| --- | --- |
| **Управление на разходите** | Свързва се с принципите на Build-Measure-Learn, ориентирани към ускоряване времето за разгръщане на нови версии. |
| **Оперативно съвършенство** | Автоматизиране на работната среда и операциите, за да се увеличи общата производителност и да се намалят човешките грешки. |
| **Ефективност** | Отговаряне на изискванията, поставени върху работни натоварвания, чрез тестове за производителност и натоварване, за да се идентифицират потенциалните затруднения. |
| **Надеждност** | Тази точка се отнася до високата наличност, която да осигурява на потребителите достъп до системата в 99.9% от случайте. Приложенията на клиентите трябва да очакват неуспехи и да се възстановят от тях. |
| **Сигурност** | Специално внимание очаква да се обърне на управлението на идентичността, достъпа до инфраструктурата, сигурността на приложенията и криптирането на данни. |

Изхождайки от казаното до тук, може да обобщим, че облачните изчисления осигурят инфраструктура за приложения, предлагайки ресурси като сървъри, операционни системи, защитни стени, балансьори на натоварването и много други. Хардуерът е разположени в център за данни, поддържан от облачен доставчик като Microsoft, Amazon, Google или друг, като се гарантира, че нужните инструменти са налични. Идеята е ИТ специалистите да могат като потребители да създават виртуални ресурси по всяко време, без необходимост от закупуване или поддръжка на устройства. Представено на фигура 1.Х, традиционния подход, при който хардуера е служебен и изисква цялостно управление и поддържка от ИТ отдела. Противоположно на това, инфраструктурата като услуга (IaaS) абстрахира физическия хардуер в среда, управлявана външно, позволявайки на организациите да изнесат слоевете на мрежова връзка, съхранение и виртуализация, като същевременно запазват контрол върху операционните системи и приложенията. Като допълнение, платформа като услуга (PaaS) предоставя и управление на операционните системи, междинния софтуер и средите за изпълнение, като по този начин позволява на разработчиците да се съсредоточат единствено върху създаването и внедряването на приложения. Софтуерът като услуга (SaaS), най-абстрактният модел, доставя напълно функционални приложения, изцяло управлявани от доставчика на услуги. Този модел позволява разработчиците да се концентрират върху бизнес полезността на софтуера. Всеки от тези модели очертава различно ниво на контрол, сложност на управление и оперативни разходи, което ги прави подходящи за различни организационни изисквания и възможности.



Фиг. 1. X. Диаграмата сравняваща моделите на облачни изчислителни услуги (IaaS, PaaS, SaaS) и традиционната локална инфраструктура, подчертавайки отговорностите за управление. Източник: АБВ, 2023

Според статистически изследвания в АБВ, IaaS, PaaS и SaaS моделите са по-ефективни от базовият, тъй като се заплаща само за услугите, които се използват. Ценообразуване, базирано на потреблението, позволява на компаниите да плащат за ресурси само когато са необходими, което намалява първоначалните разходи и позволява по-добро прогнозиране. Също така се има възможност за увеличаване или намаляване на ресурсите въз основа на натовареността в определен момент. Друга популярна опция са т.нар. „изчисления без сървър“. Те позволяват отделни функции да се изпълняват, когато се стартира някакво действие. Въпреки че не са подходящи за всяко приложение, това е бърз вариант за внедряване, когато логиката на е разделена на независими единици или за автоматизирани задачи. За тях се заплаща на базата на отделните заявки към функцията. Също така облачните доставчици осигуряват архивиране на данни, възстановяване „след бедствие“ и услуги за репликация.

Проучвания на източници в областта показват, че за конструиране на облачни системи се препоръчва ориентирания към микроуслуги архитектурен стил (microservices). Това е подход за изграждане на сървърни приложения като набор от малки, но високо-качествени под-услуги. Съответно, клиентите, на сървърните услуги, могат да бъдат отделни приложения, които да се поддържат и управляват самостоятелно. Всяка услуга работи в собствен процес и комуникира с други процеси, използвайки различен тип и вид протоколи като: HTTP/HTTPS, WebSockets, AMQP, gRPC и мн други. Всеки микросървис притежава специфична бизнес способност и предимства като това да бъде проектирана, разработена и внедрена независимо от другите. Работата може да бъде дистрибутирана между отделни екипи, осигурявайки възможно за независима работа по отделни области на приложението. Микроуслугите насърчават фактор #6 от принципите на дванадесет-факторното приложение, който свързва притежанието на всяка услуга със своя собствена логика и данни, в рамките на автономен жизнен цикъл. Концептуалните модели, технологиите и проблемите се различават между подсистемите или микроуслугите. Този принцип е заложен в дизайнът, управляван от домейн, където всяка услуга притежава свой модел на домейн (данни + логика и поведение).

За разлика, монолитните приложения представляват традиционен модел на софтуерна архитектура, при който всички компоненти на приложението са тясно интегрирани и разгърнати като едно цяло. Тази архитектура, преобладаваща в разработката на софтуер от много години, въплъщава унифициран модел, в който различни функции – като въвеждане на данни, обработка и потребителски интерфейс – са преплетени в една програма. Въпреки присъщата простота и първоначалната лекота на разработка, свързани с монолитните архитектури, литературата по софтуерно инженерство показва склонност тези системи да се превърнат в „недодялани“ структури. В монолитните приложения всички компоненти са тясно свързани и взаимо-зависими, чиято сложност ескалира с времето и води заплетена, неструктурирана и трудна за поддържане кодова база. В този смисъл, подобренията или модификациите на една част от системата могат неволно да засегнат други несвързани секции. Следователно, отстраняването на грешки се усложнява, което от своя страна възпрепятства въвеждането на нови функционалности. Този феномен, документиран в различни казуси за разработка на софтуер, подчертава значителен недостатък на монолитните архитектури в контраст с подход към микроуслуги. Целта на микроуслуги е да се достави функционален продукт, изискващ постоянна поддръжка и тясна връзка с клиента. Това увеличава удовлетвореността, както и променя гледната точка на разработчиците.

Ако погледнем в исторически план, service-oriented architecture (SOA) често се разглежда като предшественик на архитектурата на микроуслугите. SOA възниква в началото на 2000-та година, поставяйки основата за модулен софтуерен дизайн. Традиционните реализации на SOA използват сложни механизми като Enterprise Service Buses (ESB) и протоколи като SOAP и WS-\* което е усложнено и предизвикателно за поддържане. От друга страна, микроуслуги използват протоколи, които мрежата вече предоставя (обикновено REST API), което ускорява разработката и улеснява поддръжката на приложения. С Microservices всеки екип е отговорен по отношение на платформата за разработка, базата данни и създаването на регистрационни файлове. Както пише Vogels, W (2023), който е главен технологичен директор (CTO) на Amazon Web Services: „Вие го изграждате, вие го управлявате“, имайки се в предвид, че се дава възможност вземането на оптимални технологични решения за конкретен случай на употреба.

Важно да отбележим, че достъпът до данни е по-сложен в архитектурата на микроуслуги в сравнение с монолитната. Данните, притежавани от една микроуслуга са частни и трябва да бъдат достъпвани или синхронно, чрез нейните API крайни точки или асинхронно чрез поискване на съобщение. Капсулирането на данните гарантира, че микроуслугите са слабо свързани и могат да се развиват независимо една от друга. Ако множество услуги получат достъп до едни и същи бази данни, актуализациите на схемите ще изискват координация, което би нарушило автономността на жизнения цикъл. Съгласно АБВ, когато един бизнес процес обхваща множество микроуслуги се препоръчва използването на т.н. „евентуална последователност“, което сравнение по-трудно за изпълнение от обикновените SQL съединения в монолитната архитектура. Различните микроуслуги съхраняват и обработват различни видове данни, което води до използване на смесица от бази, наричащо се подход на „полиглотна устойчивост“ (Polyglot Persistence). Както беше споменато, за някои случаи на употреба NoSQL база данни като MongoDB може да има по-удобен модел, както и да предлага по-добра производителност от SQL база данни като MSSQL.

Проучвания на източници в областта, показват че облачните системи зависят от много различни спомагателни ресурси, като хранилища за данни, брокери на съобщения, мониторинг, услуги за идентичност и механизми за кеширане. Тези услуги са известни като поддържащи услуги. Фигура 1.Х показва общи услуги за поддръжка, които използват облачните системи.



Фиг. 1.Х Спомагателни услуги използвани от облачните системи (Smith, 2023).

Според Smith (2023), облачните доставчици предлагат богат асортимент от спомагателни услуги. Облачния доставчикът управлява ресурса и носи отговорност за производителността, сигурността и поддръжката. Мониторингът, резервирането и наличността са вградени. Считаме че, добра практика за тези услуги е да се третират като прикачен ресурс, динамично свързан с главна микроуслуга чрез конфигурация (URL и идентификационни данни).

Както беше споменато по-горе, облачните системи се характеризират с **висока производителност**. Според АБВ, производителност е важен показател в сферата на софтуера, представляващ продължителността между заявката на потребителя и последващия отговор от системата. Според съшия автор, производителности служи като мярка за ефективност пряко се свързва с удовлетвореността на потребителите. Бързото време за реакция обикновено означава оптимална производителност на системата, което води до положително потребителско изживяване, докато забавянето може да е показател за неефективност. АБВ представя общ метод за концептуализиране на производителността, чрез следното уравнение:

*Време за отговор = Време за обработка + Време на изчакване*

В случая, *време за отговор* е общото време, необходимо от момента, в който потребителят изпрати заявка до момента, в който получи отговор, иначе казано това е времето, което потребителят чака да види резултат след започване на действие. *Време за обработка* е действителното време, необходимо на системата за изчисляване на резултата след получаване на заявката. То включва задачи като запитване към бази данни, обработка и всяка друга работа, която системата извършва, за да изпълни заявката. От друга страна, *време на изчакване* представлява времето, в което заявката прекарва в чакане на „опашка“, преди да бъде обработена. В система с голям трафик могат да постъпят няколко заявки едновременно. Ако системата не може да ги обработи наведнъж, някои заявки трябва да изчакат, като по този начин се увеличава времето за изчакване. Съгласно АБВ, чрез разбирането и разделянето на времето за отговор на неговите компоненти, системните администратори и разработчиците могат да определят областите за подобрение. Например, ако времето за обработка е дълго, може да е необходима оптимизация на алгоритми или код. Ако времето за изчакване е дълго, това може да е знак, че системата се нуждае от по-добро балансиране на натоварването или увеличен капацитет за обработка.

В тази връзка, много автори дават значение на **нивото на латентност**, тъй като то гарантира, че преобладаващата част от клиентите получават последователно и отзивчиво обслужване. Често в софтуерните системи 95% (означено в техническата литература като P95) от заявките се обработват в сравнително оптимално време, докато 5% отнемат повече. Днес в специалната литература, крайната латентност се свързва с ефикасността на заявките в горните проценти (като P95, P99 и P99.9), акцентирайки върху „слабите“ заявки. Голямото забавяне може да означава проблеми, които възникват само при определени условия, като конкуренция за ресурси, забавяне на работната рамка, хардуерни проблеми или други.

Според проучване на Google, 53% от мобилните потребители изоставят сайтове, зареждането на които отнема повече от 3 секунди. Вземайки предвид, че трафикът и натоварването са динамични променливи, влияещи пряко върху производителността на системата и възможността за разширяване на бизнеса, в съвременната ситуация **мащабируемостта** е възможно решение на тези проблеми. Проучвания на източници в областта, показват че способността на система да управлява ефективно увеличеното работно натоварване се отнася до мащабируемостта. Публикациите в областта, описват две измерения на мащабируемостта: вертикална и хоризонтална. Според АБВ, вертикална мащабируемост се свързва с надграждане на физическия хардуер като процесор, памет или честотна лента на мрежата. За сметка на това, хоризонтална мащабируемост се постига чрез добавяне на ресурсни единици. Вместо да се подобрява един сървър, множество виртуални сървъри се създават, за да се разпредели натоварването. Този подход може да предложи висока достъпност и толерантност към грешки, но същевременно въвежда сложност при координацията между ресурсите.

Изхождайки от казаното до тук, може да обобщим че **висока наличност** е първостепенен атрибут на качество, предоставен от облачните доставчици. В изследваната от нас литература, наличността на система се определя като частта от времето, през което дадена услуга е функционална и достъпна. Според АБВ, наличността може да бъде изразена като процент от времето на работа (uptime) спрямо сумата от времето на работа и времето на в застой (downtime):

*Availability = uptime / (uptime + downtime)*

Счита се, че за повечето облачни услуги абсолютната 100% наличност е нереалистична поради необходимостта от поддръжка и надстройки. Статистически, 90% наличност се равнява на над 2 часа ежедневен застой или 36 дни годишно. 95% се равнява на около час дневно или 18 дни годишно, в които системата е офлайн. Индустриалните стандарти, определени от облачните доставчици, се движат около 99% до обикновено 99,9% (наричани „три деветки“), представлявайки застой от по-малко от 1,5 минути дневно.

В тази връзка, трябва да отбележим **споразуменията за ниво на обслужване** (SLA), което представляват договори между облачните доставчици и предприятията. SLA въплъщават обещанията за по-горе посочените наличност, производителност, нивото на латентност и време за реакция. Неспазването на SLA от доставчиците на услуги може да доведе до различни последици, включително правни и договорни санкции. От друга страна, индивидуалните цели, определени за една система се наричат цел за ниво на обслужване (SLO). Всеки SLO показва целева стойност или диапазон за специфични системни аспекти, като време за реакция под 100 ms на 90-ия процент. В тази връзка, индикаторът за ниво на обслужване (SLI) е количествена мярка за определяне на спазването на SLO. Той представлява данните за ефективността в реално време, които се събират и оценяват дали се постигат SLO. Според нас SLA, SLO и SLI са основни за осигуряване на качество на облачна услуга. Докато SLA често се определят от юридически екипи, SLO и SLI са повече в областта на софтуерните архитекти и инженер.

## 1.3. Управление на бизнес процесите чрез ориентиран към домейн дизайн

Уеб услугата, независимо дали става въпрос за монолитна система или част от разпределена инфраструктура, представлява комплекс от характеристики, които включват обема на обработваната информация, ефективност, бизнес логика и технологично развитие (AБВ, ). В изследваните от нас литературни и интернет източници свързани с „ориентиран към домейн дизайн“ (DDD) се подчертава ефективността му в ситуации, при които има нужда от обработка на сложни бизнес изисквания. С други думи, DDD цели да се справи със сложността на домейна, която обхваща бизнес правила, валидации и изчисления. Изхождайки от определенията и предложенията към от глава 1.1. управление на поръчките от клиенти в производствено предприятие се отличава със сравнително висока сложност и необходимост от изпълнение на поредица от процеси.

Класическият подход, описан от Т. Ерл в книгата „*Принципи на дизайна на ориентирана към услуги архитектура“*, определя разделянето на услугите въз основа на данните, с които функционалностите им оперират. Този подход е познат като „*дизайн, управляван от данни*“ (data driven design). От друга страна, Е. Еванс твърди, че DDD предоставя ключови концепции за разделяне на уеб услугите, на базата на други фактори. Формулирайки методология, DDD предоставя начин за представяне на реалния свят чрез структурирано решение, което отговаря на изискванията в проблемното пространство. Тези характеристики допринасят за подобрение на качеството на софтуерната архитектура.

В този смисъл, сложността на бизнес логиката представлява индикатор за сложността на проблемната област, която софтуерът цели да реши. Ако приложение, извършва основни операции като създаване, четене, актуализиране и изтриване (CRUD), може да обобщим че то не би съдържало сложна логика и може да бъде реализирано с по-прости методи от DDD. От друга страна, система за управление на поръчки, която автоматизира голяма част от операциите на произведена компанията и моделира голяма част от логистичните процеси, следователно управлява множество сложни бизнес задачи. Счита се, че тази сложност, отнасяща се до броя на алгоритмите и технологиите, които трябва да бъдат внедрени, за да се осигури правилната функционалност на софтуера, може да бъде много висока.

В книгата си „*Patterns of Enterprise Application Architecture*“, Мартин Фаулър представя диаграма (фигура 1.Х), която илюстрира връзката между времето, цената и сложността при проектирането на софтуер. В тази диаграма по оста Y са представени времето и цената, докато по оста X е измерена сложността на проекта. Този модел е свързан с ориентирания към данни подход за проектиране на софтуер и показва, че при този метод, след достигане на определено ниво на сложност, дори незначително увеличение на сложността може да доведе до значително увеличение на разходите и времето, необходимо за разработката.



*Фигура 1.Х. Домейн-центрирано срещу данни-центрично в контекста на диаграма за разработка на софтуер, изобразяваща време и сложност.*

Срещу това, при ориентирания към домейна подход времето и разходите за проекта имат тенденция да нарастват линейно, като началните разходи обаче биват по-високи. Според принципите на DDD, случаите на употреба следва да се моделират въз основа на начина, по който реалният бизнес функционира, като взема предвид, че този той постоянно се развива.

DDD предоставя различни технически концепции и модели, които могат да бъдат използвани за внедряването на софтуерни проекти. Считаме че, представени като контекстна карта на фиг.1.Х се улеснява идентифицирането и управлението на взаимозависимостите и сътрудничеството помежду им. Картата има за цел да даде структура на облачната система, както и да бъде пътеводител в по-голямата картина.



*Фигура 1.Х. Карта, описаваща връзките в DDD.*

Тези концепции включват универсален език (UL), ограничен контекст (BC), агрегати, обекти на основния домейн, стойностни обекти и хранилища на данни и много други. Въпреки че, тези технически аспекти са много на брой и според критици са трудни за научаване, те са част от правилното прилагане на DDD методологията.

В различните индустрии се използва специфична терминология, която отразява определен бизнес контекст. В този смисъл, когато се разработва сложна система за управление, е важно да се разбере и използва терминологията, както и да се осигури нейно съответствие в програмния код, за се реализират бизнес целите. Основна характеристика на DDD е улесняване на комуникацията между експертите по домейна и софтуерните инженери, като се дефинира общ, **универсален език** (UL). Това е инструмент, който помага на обединяването на бизнесът, дизайнерите и програмистите, така че те да могат да създадат модели на домейна и да ги приложат в практиката. Когато кодът е написан на UL, той може да даде подсказки за случаи и изисквания, които не са били достатъчно ясни предварително. За да функционира успешно, класовете в кода и таблиците в базата данни трябва да се именуват в съответствие с термините от UL. Тази обща номенклатура улеснява разбирането и съгласуването на изискванията между всички заинтересовани страни. В своето изследване АБВ, подчертава важността на универсалния език за предотвратяване на недоразумения и неправилни предположения. UL се използва в различни аспекти на разработката на софтуер, включително в документацията, комуникацията между екипите, кода на приложението и кода за тестване. Този универсален език се развива и се поддържа с течение на времето, като предоставя средство за събиране и организиране на знанията и бизнес логиката.

В този смисъл, **ограниченият контекст** (BC) е малка област в домейна, която дава на всеки елемент от UL собствено значение. Според практици в областта, често кодовата база на приложение става неуправляема, когато обемът се увеличи. BC контролира как са структурирани подпрограмите и тяхното развитие. Често BC съответства на под-домейн, който показва как е разделена дейността на бизнеса. Всеки BC се разработва самостоятелно, като може да бъде една или няколко микроуслуги.

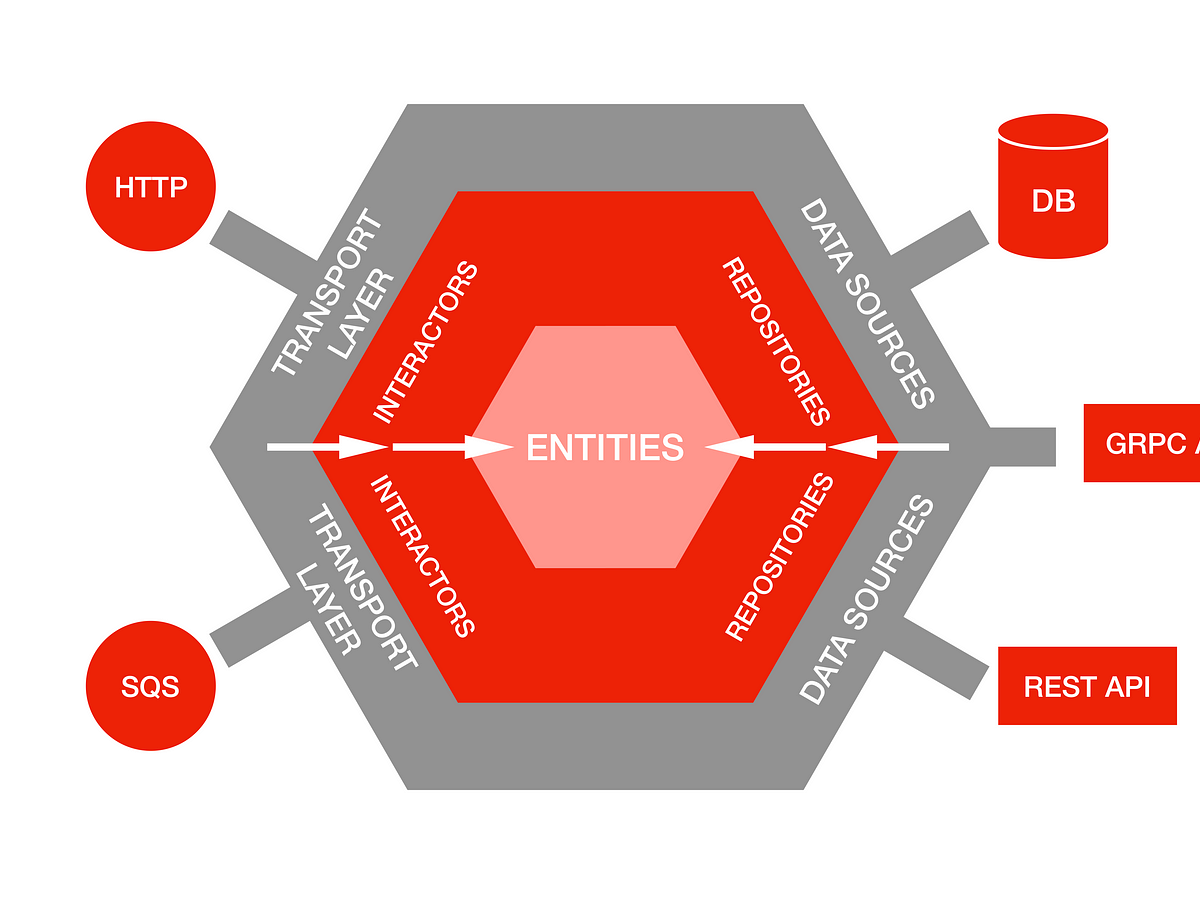
В книгата си Vaughn Vernon (2013) казва, че т.н. **стойностни обект**и следва да са малки, прости обекти, които не се основават на идентичност. Те са елементи, използвани за количествено определяне, измерване или характеризиране на определена тема. Стойностните обекти могат да имат методи и поведение, но никога не трябва да имат странични ефекти. Примери за такива обекти могат да бъдат представените по-горе организационни структури.

От друга страна, сред публикациите в DDD областта, **агрегатът** е представен като колекция от свързани елементи, които се модифицират като едно цяло. В този смисъл, агрегатите се третират като единица за промени в данните. Те се състоят от един или повече обекти, които се променят заедно. Преди да се направят модификации, е необходимо да се оцени консистенцията на целия агрегат, като може да има правила, които да гарантират, че всички данни на обектите са последователни. Промените в данните на агрегатите следва да бъдат атомарни, последователни, изолирани и дълготрайни (ACID).

Според АБВ, **хранилищата за данни** са колекция от елементи от определен тип. Те предлагат унифицирана абстракция за всички проблеми, свързани с четенето и записването на данни, като улесняват програмистите, които ще получават и управляват моделни обекти. Хранилищата предоставят и регулират достъп до базата с данни, чрез публичният си интерфейс. Друго предимство е, че те правят кода по-лесен за тестване в сравнение традиционния вариант, при който външни ресурси се свързва пряко. В този смисъл, тъй като кодът за достъп до данни е обвит в един или повече класове, той е по-лесен и по-безопасен за използване.

Vaughn Vernon описва **събития в домейна**, като казва, че те трябва да се използват за улавяне на събитие, което се е случило в домейна, и трябва да бъдат част от UL. Събитията са полезни, защото сигнализират, че определено нещо се е случило. Домейн събитието е по същество съобщение, запис на нещо, което се е случило в миналото.

Според практици в областта на софтуерното инженерство **Hexagonal, Clean, and Onion** архитектурите поддържат високи нива на модулност и разделяне на проблемите. Hexagonal (или шестоъгълна) архитектура, представена на фиг.1.Х поставя модел, при който ядрото на приложението е отделено от външни системи чрез дефинирани портове и адаптери, като по този начин улеснява взаимозаменяемостта и тестването.



*Фигура 1.Х. Модел на Hexagonal архитектурa*

Clean (или чиста) архитектура, представена от Robert C. Martin и илюстрирана на фиг 1.Х, се базира на Hexagonal, като я допълва чрез разделяне на системата на слоеве с правила за зависимости, което гарантира, че вътрешните слоеве остават незасегнати от промените във външните.



*Фигура 1.Х. Модел на Clean архитектурa*

Комбинацията от разгледаните две архитектури с DDD принципите, описани го горе, централизира модела на домейна, обкръжавайки го с приложни и инфраструктурни слоеве. Представена на фиг. 1.Х onion архитектура използва тези слоеве и централно ядро. Горните слоеве зависят от долните, но не и обратно, показвайки, че основните елементи на DDD трябва да работят независимо един от друг.



*Фигура 1.Х.* *DDD in onion architecture*

В този смисъл, важен аспект от проектирането и създаването на услуга е определянето на нейните граници. Както беше споменато по-горе, всеки BC идентифицира субектите и стойностните обекти, характеризира ги и ги комбинира. Изборът къде да се направи границата между BC изисква балансиране на две конкуриращи се цели. Създаването на бариера около елементите, които се нуждаят от сплотеност, е първата. Втората цел е да се избегнат „бъбриви“ комуникации между звената. Тези цели могат да противоречат една на друга. Балансът трябва да се постигне чрез разлагане на системата на възможно най-малките единици. Друг начин този аспект да се разгледа е автономността. Една работна единица не е напълно автономна, ако разчита на друга, за да изпълни заявка.

На базата на направени проучвания, голямата част от корпоративните приложения имат различни нива, които помагат на разработчиците да управляват сложността на кода. Спазвайки принципите на DDD, елементите могат да бъдат организирани в няколко слоя, както е показано на фигура 1.Х.

Diagram

Description automatically generated

*Фигура 1.Х.* *Трислоен архитектурен модел, който се използва за разделяне на проблемите и подобряване на управляемостта.*

Счита се, че приложният слой координира потока на изпълнение между различни обекти. Той определя случаите на използване и операциите, които могат да бъдат извършени в рамките на услуга като организира взаимодействието между потребителския интерфейс и основните елементи. Обикновено приложният слой се реализира като уеб API или MVC проект. Приложният слой е зависим от другите два. От друга страна, слоят на домейна капсулира бизнес логиката и основните обекти, които съставляват ядрото на услугата (агрегати, стойностни обекти и др). Той се концентрира върху решаването на бизнес проблеми и изразява концепциите и поведението на бизнес домейна. От гледна точка на кода, този слой следва да има напълно отделени класови обекти, за да не зависи от никой друг. За сметка на това, инфраструктурният слой е отговорен за осигуряването на необходимите технически инструменти. Основната му функция е да абстрахира и капсулира технически подробности. Той предоставя реализации за множество проблеми, включително запис на данни, съобщения, мрежова комуникация, интеграция с външни услуги, кеширане и оптимизиране други.

В този смисъл, Грег Йънг представя концепцията за разделяне на отговорността за команди и заявки (CQRS) през 2010 година като разширение на принципите на DDD. Тази идея се базира на принципа на Bertrand Meyer, наречен "разделяне на команди и заявки" (CQS). Съгласно този принцип, всеки метод в API трябва да бъде или команда (command) или заявка (query), но не и двете едновременно. Според Йънг, командите са методи, които извършват операции, променящи състоянието на системата. Те са отговорни за изпълнение на действия, които променят данни или файлове. Заявките, от друга страна, се използват за извличане на информация, като предоставят данни, но без да променят стойностите.

Един от аспектите на CQS е, че методите би трябвало да връщат стойност само ако са "референтно прозрачни" и нямат "странични ефекти, което прави кода по-четлив и предсказуем. Въпреки това, не винаги е възможно или практично да се съобразяваме стриктно с принципа на CQS. Има сценарии, когато методите трябва да имат както страничен ефект (промяна на състоянието), така и да връщат стойност. Например, при работа със структура от данни "Стек", методът "Pop" премахва и връща последния елемент от стека (Наков, 2022). В този случай разделението на тези задачи на два отделни метода може да стане нелогично. Следователно, важно да се анализират конкретните изисквания и сценарии на приложението, преди да се прилага стриктната парадигма на CQS. В тази връзката, вместо да се фокусира върху методи като CQS, CQRS прилага същите принципи, като се насочва към разделяне на операции: една за управление на записите (командите), а другата за обработка на четенето (заявките). Считаме, че чрез това разделение може да се разработят различни стратегии, които да се фокусират върху конкретните нужди на облачно базираната система. Приложният слой преобразува входните заявки и команда и ги изпраща по споделен комуникационен канал, известен като „манипулатор на съобщения“. В този контекст, командите се използват за да кажат на приложението да извърши определено действие, заявките се използват за да поискат информация или данни от приложението, а събитията представляват информационни съобщения. Командите предизвикват реакции в модела на домейна, а събитията са резултат от тях. Именуването на съобщенията следва стандартизирани указания на UL, като командите винаги са в повелително време, заявките обикновено започват с "GET," а събитията винаги са в минало време.

Според АБВ, теоремата на CAP (или теоремата на Брюър) е основен принцип в областта на разпределените системи, която има тясна връзка с CQRS. Съгласно CAP, че разпределена система не може да гарантира едновременно всички три от следните възможности:

1.Последователност (Consistency): Всички операции на четене връщат запис или грешка;

2.Достъпност (Availability): Всяка заявка получава отговор, дори ако не всички части от системата са достъпни;

3.Производителност (Partition tolerance): Системата продължава да работи дори при загуба или забавяне на комуникацията между различни части в мрежата;

Считаме, че чрез внедряването на CQRS, разработчиците могат да създават облачни услуги, които се справят ефективно с големи натоварвания от заявки, като същевременно гарантират съгласуваност на данните чрез стриктната обработка на командите. CQRS обикновено се използва като междинен етап преди източника на събитие. Извличането на събития допълва CQRS, като събира всички промени в състоянието на системата като поредица от събития, които могат да бъдат използвани за съгласуване и анализ на данните.

Извличането на събития (ES) е техника за проектиране, базирана на концепция, че всички промени в състоянието на приложението, през целия му жизнен цикъл, се записват като поредица от събития. В резултат на това, събитията (events) се превръщат в основен градивен елемент на приложението. При подхода за източник на събития програмите съхраняват транзакции, но не и съответните им състояния. Когато е необходимо да се изтегли текущото състояние от базата, следва да се приложат всички транзакции от началото. Нищо не се изтрива или променя в хранилището за данни. Поради това, не може да се срещне проблеми с едновременната актуализация. На базата на направени проучвания се установи, че повечето приложения работят като съхраняват текущото състояние. Вместо да съхранява цялата информация в колоните на един запис или в свойствата на един обект, чрез ES състоянието на обектите се описва от последователността от събития. Това е т.н. *представяне на субект, базирано на събития*. За да се получи текущото състояние на определенa същност, необходимо е да се повтори времевата линия на програмата от самото начало. Този алгоритъмът включва изследване на данните и използване на логика за извличане на съответната информация. Чрез използването на записаните събития, е възможно да се реконструира състоянието на определен агрегат. Това понякога може да изисква управление на огромни обеми от данни. В такъв случай могат да бъдат записани проекции, които представят състоянието в определен момент от време. Веднъж съхранени, събитията са неизменни. Изхождайки от теоретичните постановки, свързани с концепцията, считаме че, съхранението на събития може да бъде релационно, базирано на документи или базирано на графи, следователно събитията могат да се съхраняват в SQL или NoSQL база данни като PostgreSQL, MySQL, MongoDB, Apache Cassandra, или могат да се съхраняват с помощта на по-специфични решение като „RavenDB“ или „FaunaDB“.

В този смисъл, разработка, управлявана от тестове (TDD) и DDD са две мощни методологии, които, когато се комбинират, могат да повишат качеството на облачните услуги и процеса на разработка. Използвайки тези практики, разработчиците и инженерите за осигуряване на качеството могат да създадат система, която е по-здрава и надеждна. TDD насърчава строг процес на тестване, при който тестовете се пишат преди кода за внедряване; този процес следва добрите практики, като гарантира, че предвидената функционалност е изпълнена. Съществува процедура в три стъпки, известна като „red, green, refactoring“. Създаването на неуспешен тест за част от функционалността е първата стъпка. Втората фаза е „зелената стъпка“, по време на която се създава достатъчен производствен код, за да премине неуспешният тест. “Рефакторингът” е последната фаза, в която тестовият и производственият код се подобряват, за да се поддържа високо качество. Този цикъл се повтаря за всяка част от функционалността в реда на нарастване на сложността във всеки метод и клас, докато не бъде завършена цялата функция. Използването на TDD гарантира, че процесът на тестване е това, което ръководи дизайна.

В областта на софтуерното тестване съществуват няколко различни вида. Някои тестове са базирани на предмета – например тестване на единица, интеграция, услуга на компонент и потребителски интерфейс. Междувременно други се определят от целта на теста – например функционални тестове, тестове за приемане, тестове за дим и проучвателни тестове. Други пък се определят от това как се тестват – например автоматизирани, полу-автоматизирани и ръчни тестове.

Пирамидата за автоматизация на тестовете (фиг.1.Х) е представена за първи път от Майк Кон в книгата му Succeeding with Agile: Software Development Using Scrum. Пирамидата изобразява видовете тестове, които трябва да се извършват на различни етапи от жизнения цикъл на разработка на софтуер и колко често следва да се адаптират в тестов пакет, за да се гарантира качеството на програма. Идеята зад пирамидата е, че тестерите трябва да посветят повече усилия на основните тестове, преди да преминат към по-сложни.

Diagram

Description automatically generated

*Фиг. 1.Х. The agile test automation pyramid by Mike Cohn*

На фигура 1.Х са идентифицирани четири различни вида тестове:

1) Единични тестове - автоматизирани тестове, които проверяват колко добре работи отделна част от кода сама по себе си;

2) Сервизни тестове - автоматизирани тестове, които проверяват колко добре работи група от класове и методи, които предоставят услуга на потребителите;

3) UI тестове - автоматизирани тестове, които проверяват дали цялото приложение работи (от потребителския интерфейс до базата данни);

4) Ръчни тестове - тестове, извършвани от лице, което също така проверява пълната функционалност на приложението;

Счита се, че чрез внедряването на TDD програмистите имат способността да идентифицират потенциални проблеми на ранен. В допълнение, итеративният характер на TDD позволява честа обратна връзка, което улеснява непрекъснатото усъвършенстване и адаптивност при разработването на облачни услуги.

Важно да се отбележи, е че техниките, изложени в тази подглава, не са подходящи за всички ситуации и следователно имат някои ограничения, като необходимостта от време и усилия за правилното разбиране и прилагане на многобройните DDD слоеве, модели и концепции. Според литературните източници, кривата на обучение за DDD е „стръмна“. В този смисъл, дизайнът на всеки елемент в системата показва свои собствени компромиси и вътрешни решения.

Като заключение, подходите за проектиране, управлявани от домейн, се очертават като методология за изграждане на архитектури на облачни услуги. Капсулирането на основния бизнес домейн в добре дефинирани, ограничени контексти, помага за правилното създаване на подсистеми, модули и обекти. Чрез комбиниране на споменатите подходи производствените организации могат да изградят системи, които са не само технически стабилни, но и съобразени с бизнес целите и изисквания. В крайна сметка, възприемането на управлявания от домейн дизайн и облачни архитектури следва да помогне на организациите с внедряването на иновации, намаленото на разходи, предоставянето на услуги на своите клиенти и конкурентоспособността в бързо променящия се дигитален пейзаж. В тази връзка, неспособността да се поддържа адекватно разделяне на проблемите в корпоративния софтуер е основна причина за претоварени кодови бази, което от своя страна води до забавяния и дори провал на проекти. Тъй като тази част се фокусира най-вече върху теоретичните основи, като продължение във втора и трета глава предстои да се представи практически ориентиран казус върху фактическата реализация.Тази глава разглежда решение от високо ниво, което да съсредоточава върху всички основни потребителски, бизнес и ИТ изисквания. Важна част от глава 2 са градивните елементи и интерфейси, изграждащи системата, както и комуникационните модели, които да ръководят композицията. Сложността на операциите стреми да бъде сведена до минимум. Представени са всички случаи на употреба и бизнес сценарии, съвместно с които се моделират приложенията за обслужване на клиенти. Освен това дизайнът обхваща функционалност, използваемост, устойчивост, производителност, икономически, технологични ограничения, компромиси и естетически проблеми на клиентските и сървърни приложения.

# Глава 3. Изграждане и използване на облачна система за производствено предприятие "Хейделберг Цимент Девня" АД

## 3.1. Обща характеристика на дейността на компанията

"Хейделберг Цимент Девня" АД е най-големият производител на цимент в България, разположен в град Девня, област Варна, в експлоатация от 4 декември 1958 г. "Хейделберг Цимент Девня" АД е част от Heidelberg Materials, основан през 1874 г., който е основен участник в глобалната индустрия за строителни материали. Основната дейност включва производство и дистрибуция на цимент, инертни материали, готови бетонови смеси и асфалт. Компания е специализирана в производството и доставката на бетонова смес, която се произвежда в централно съоръжение за дозиране. Терминът „готови смеси“ произлиза от факта, че тези смеси се произвеждат според спецификациите на клиента, което води до прецизен, висококачествен продукт, който може да се използва веднага след доставката. За дозиране на бетон, компанията използва големи централни съоръжения. Това дава възможност за прецизно и точно смесване, гарантирайки целостта на продукта. "Девня Цимент" произвежда смеси според изискванията за здравина, обработваемост и издръжливост. Често готовата смес се доставя в камиони с миксери и трябва да се използва веднага след пристигането. В тази връзка, проследяването на точната му местоположение е от съществено значение. Също така, превозните средства имат сензори, които изпращат информация за нивото на водата, градусите и други характеристики на сместа в реално време, за да се гарантира високо качество на продукта, тъй като свойствата му могат да се променят междувременно. Счита се, че централизираното смесване е по-благоприятно за околната среда от смесването на работната площадка, тъй като отпадъчния продукт е по-малко, а същевременно позволява по-голям контрол върху използваните материали. В този смисъл, продуктите на компанията се използват за изграждане на къщи, инфраструктура, търговски и промишлени съоръжения, като по този начин отговарят на нуждите на нарастващото световно население за жилища, мобилност и икономическо развитие.

Въз основа на анализ на данните от проведените от нас интервюта с специалисти в областта, могат да се формулират следните бизнес процеси:

* Приемане на поръчка: Считаме, че това е първият етап, който нашата система обхваща. Към момента, поръчките се получават по имейл или телефон и се обработват от диспечери. Това включва разбиране на нуждите на клиента по отношение на типа бетон, обема и времето за доставка, на базата на които се сключва договор. Облачната платформа следва да внедри тези процеси, чрез функциите за онлайн регистрация, съгласяване с общите правила и одобрение от диспечера. След това потребителите могат да регистрират поръчка, както и да променят или отхвърлят съществуваща;
* График: След като поръчката бъде приета, следва да бъде планирана за производство и доставка. Различните специалисти дават различни фактори, включващи производствения капацитет, управление на автопарка, периоди за доставка. Според нас онлайн порталът бива инструментът, който следва да помогне в управлението и автоматизацията на тези задачи, интегрирайки вътрешните системи с облачните услуги;
* Товарене: В този процес бетонът действително се произвежда или „дозира“. Това включва точно измерване и комбиниране на суровините: цимент, инертни материали, вода и други добавки в съответствие с конкретните изисквания. След това сместа се зарежда в превозно средство т.н. „миксер“. Според експерти, поддържането на „интелигентни“ сензори, които да изпращат данни в реално време, е пример за внедряването на иновативни идеи. Същевременно, използването на сензорите за определяне на емисиите на въглероден диоксид, повишава прозрачността в съответствие с тенденциите в индустрията към устойчиви практики, изпълнявайки регулаторните изисквания за околната среда;
* Доставка: Въз основа на стандартите ISO/IEC 27001:2013, ISO 9001:2015 и ISO 28000:2007 проследяването на доставката в реално време е основна функция. Интегрирана в облачно базирана система, тя дава възможност на шофьорите да избират оптимални маршрути, като по този начин повишава ефективността и осигурява навременна доставка. Едновременно с това улеснява непрекъснатата комуникация с клиентите, като се визуализира текущото местоположение на доставчика. След доставката, системата рационализира процеса, като позволява на клиентите да подпишат цифрово и да получат електронно доказателство за доставка, елиминирайки необходимостта от традиционни методи на хартия. Както беше описано по-горе, управлението става чрез мобилно приложение, което пази цялата информация. На работната площадката бетонът се разтоварва и поставя според изискванията;
* Фактуриране: След доставка, на клиента се издава фактура, в която е посочена стойността с начислените данъци и такси. За да се подобри бъдещото обслужване, е важно получаването на обратна връзка от контрагентите (клиенти и доставчици). Фактуриране и извършване на плащане са функционалности, които могат да бъдат интегрирани като допълнение към системата;

Според нас, ефикасността и ефективността на тези процеси пряко влияят върху качеството на предлаганите продукти и услуги, както и на рентабилността на операциите. Следователно, доброто управление е от важно значение за успеха на "Хейделберг Цимент Девня" АД.

## 3.2. Избор на технологични средства за реализация на системата

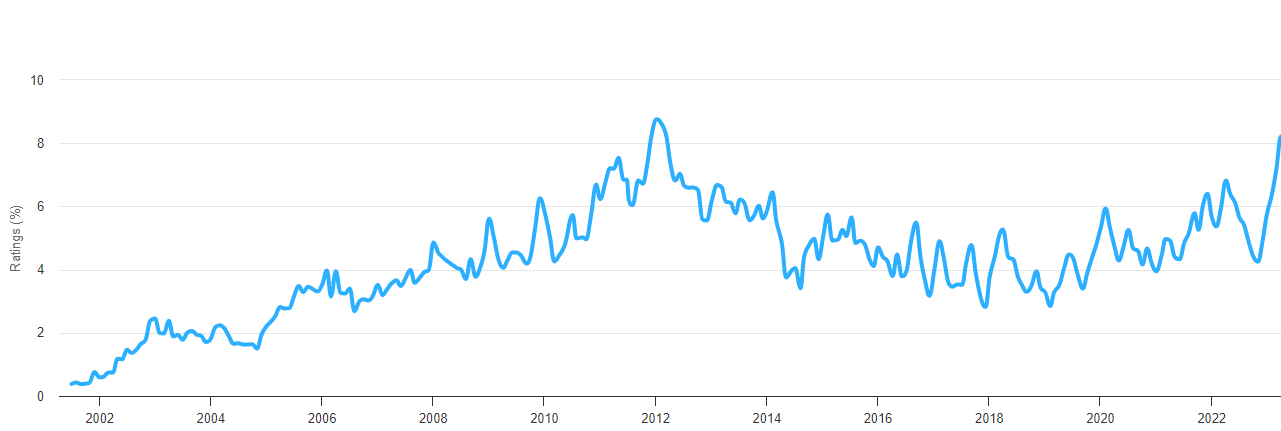
Изборът на правилните средства за реализация следва да бъде резултат от анализ и оценка на няколко технологични елемента, включващи езици за програмиране, работни рамки, бази данни и доставчици на облачни услуги. Чрез избора на технически инструменти, "Хейделберг Цимент Девня" АД следва да подобри производителността, надеждността и ефективността на процесите по управление на поръчки. Това от своя страна би трябвало да помогне в непрекъснато променящата се област на управление на веригата за доставки.

В този смисъл, таблица 3.1. представя резултати от сравнителен анализ за различни уеб базирани работни рамки, описвайки тяхната производителност.

Таблица 3.1.  
 Сравнение на сървърни технологии за разработка  
(Източник: Тechempower, 29.09.2023)

| **Сървърна Технология** | **Брой на едновременни HTTP отговори за секунда** |
| --- | --- |
| ASP .NET Core | ~ 300 613 |
| NodeJS | ~ 125 743 |
| Gin | ~ 102 559 |
| Symphony | ~ 70 382 |
| Spring | ~ 30 891 |

Данните на Techempower показват, че ASP.NET демонстрира по-висока ефективност и производителност в сравнение с други алтернативни платформи за уеб приложения, като се отбелязва, че ASP.NET Core е поне два пъти по-бърз от Node.js, който е втори в класацията. В допълнение на това, всяка година от Microsoft публикуват план за предстоящ напредък и поддръжка на .NET, осигурявайки периодични подобрения. В този смисъл, езици за програмиране C#, F#, и VB, част от еко системата на .NET и Microsoft, споделят индекси в статистиката от Tiobe и Statista. Пример е линейна графика, представена на фиг. 3.1, която очертава оценките на езика за програмиране C#, илюстрирайки тенденция на възход към края на 2023 г.



***Фиг 3.1.****: Оценки на индекса TIOBE за езика за програмиране C#* Източник: Tiobe <https://www.tiobe.com/tiobe-index/> [20.12.2023]

Освен това, GitHub дава информация за над 5,7 милиона месечно активни разработчици по проекти с „отворен код“. От своя страна, Stack Overflow отбелязва .NET Core като „#1 работна рамка“ за годините от 2019 до 2021.

В този смисъл, Microsoft Azure, доставчик на облачни услуги, предлага обширна поддръжка за .NET приложения чрез интегрираната среда за разработка Visual Studio (IDE). Според практици в областта, интеграцията от високо ниво на Azure и .NET подобрява разработката на софтуер и гарантира оперативна съвместимост в рамките на екосистема, поддържана от технологичната корпорация Microsoft. Фигура 3.2, получена от „Доклада за състоянието на облака на Flexera за 2023 г.,“ показва тенденциите за използване на различни доставчици на публичен облак в различни предприятия.



***Фиг 3.2.****: Доставчици на облачни услуги, използвани от предприятия* Източник: Flexera State of the Cloud Report [01.12.2023]

Констатациите, получени от извадка от 750 участници, показват, че над 40% от фирмите използват Azure като основна облачна платформа. Като допълнение, данните, представени от Gather за 2023 г, показват темп на растеж от 47% в облачната инфраструктура и платформени услуги, установявайки позицията на Azure като водеща публична облачна платформа.



***Фиг 3.3.****: Категоризираща на публичните облачните компании.* Източник: Gartner Magic Quadrant for Cloud Platforms [01.12.2023]

Според статистически данни, Azure разполага с мрежа от 64 центъра за данни, което надминава броя на другите облачни доставчици. Според данните от проучването се предвижда значителен ръст на приходите на Azure от около 26%, до края на 2024 г., достигайки 70 милиарда щатски долара. Някои от производствените компании, които са клиенти на Azure включват Samsung, Boeing, BMW и много други.

Въз основа на събраните данни може да заключим, че използването на .NET и Azure е благоприятен избор за внедряването на облачно базирана система за управление на поръчките от клиенти в рамките на производствена компания.

## 3.3. Физическа реализация на системата

За изграждане, доставка и изпълнение на системи, изградени както като монолитни приложения, така и като ориентирани към микро-услуги, експерти в областта препоръчват използването на контейнеризирани технологии. Според литературни източници, **контейнеризацията** е подход, в сферата на разработката на софтуер, при който кодът на приложение, всички негови зависимости и конфигурации са пакетирани в двоичен файл, наречен **изображение**. По документация, изображенията се съхраняват в **регистър**, който работи като хранилище или библиотека. Облачната платформа трансформира изображението в работещ екземпляр на **контейнер**, който може да се стартира, спира, премества или изтрива. Създават се контейнери за различните части от приложението: уеб услуга, база данни, кеширане и др. Точно както транспортните контейнери позволяват транспортирането на стоки, независимо от товарите вътре, софтуерните контейнери се възприемат като стандартна единица за внедряване на софтуер, която може да съдържа различен код и зависимости. Контейнеризирането на софтуера дава възможност на разработчиците и ИТ специалистите автоматично да подновяват новите промени в различни среди. Контейнерите също така изолират приложенията едно от друго, в споделена операционна система. Според документацията, приложенията се изпълняват върху хостът на контейнерите. В тази връзка, контейнерите предлагат предимства на изолация, преносимост, гъвкавост и контрол на целия жизнения цикъл на приложението. Според експерти в областта, най-използваната и наложила се като стандарт технология е **Docker**. Това е проект с отворен код за автоматизиране на внедряването на приложения като преносими, самодостатъчни контейнери, които работят еднакво както локално така и в облака. Docker контейнерите могат да работят върху Linux или Windows, като на фиг. 3.4. е представено сравнение между компонентите на традиционна виртуална машина и Docker контейнер. Докер контейнерите включват приложението и всички негови зависимости. Те обаче споделят ядрото на ОС с други контейнери, изпълняващи се като изолирани процеси в пространство на хост операционната система, с изключение на Hyper-V контейнери, където всеки контейнер работи вътре в специална виртуална машина. Graphical user interface

Description automatically generated

Фиг. 3.4. Сравнение между Docker и типична виртуална машина. Източник: Docker, Inc. < https://www.docker.com/>, [09.10.2023]

Според техническата литература, виртуалните машини имат три основни слоя: инфраструктура, хост, операционна система, Hypervisor и всички необходими библиотеки. Слоевете в Docker са инфраструктурата, ОС и двигател за контейнери, който поддържа изолация, но споделя основните услуги на ОС. Тъй като контейнерите изискват много по-малко ресурси (например не се нуждаят от пълна ОС), те са лесни за изпълнение, като същевременно гарантират еднакво поведение на всички среди: локална, среда за разработка или продуктивна.

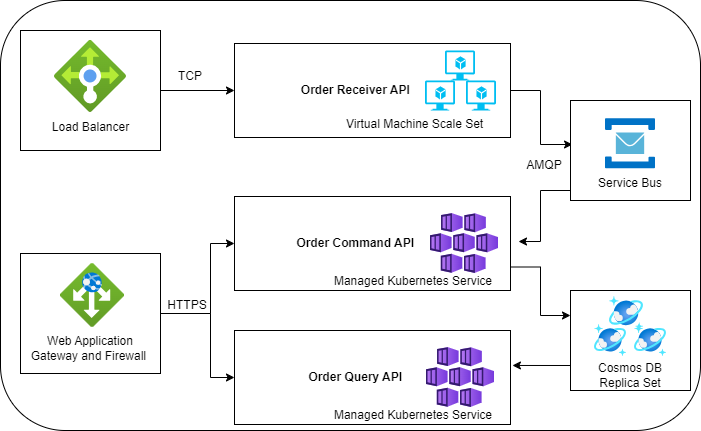
В книгата Cloud Native Patterns, авторът Корнелия Дейвис (2019) отбелязва, че контейнерите са инструмент на облачния софтуер, чието управление се извършва със специална софтуерна програма, наречена „оркестратор“. В тази връзка, следващата таблица обобщава задачи към оркестратора, описани в теорията и често срещани в практиката.

Таблица 3.3.  
 Обобщение на практиките за управление на контейнерите   
(разработка на автора)

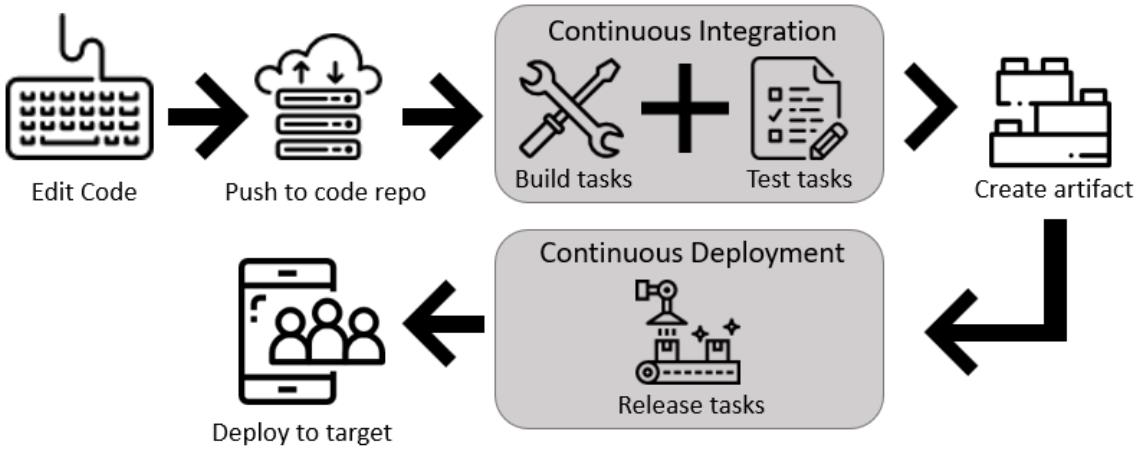
| **Задачи** | **Описание** |
| --- | --- |
| Планиране | Автоматично предоставяне на екземпляри на контейнери. |
| Мониторинг на активността | Автоматично откриване и коригиране на повреди. |
| Failover | Повторно публикуване на неуспешен екземпляр. |
| Мащабиране | Автоматично добавяне или премахване на екземпляр на контейнер, като отговор на повишен трафик. |

Според сравнителен анализ на инструменти за оркестрация, Kubernetes, проект за управление на работни натоварвания с отворен код, се очертава като предпочитан избор, главно поради цялостната си екосистема. За разлика от Docker Swarm и Apache Mesos, Kubernetes позволява автоматизация от най-високо ниво, чрез разпределяне на микроуслуги, планиране на контейнери в клъстер, предоставяне на възможности за автоматично рестартиране, повторно планиране и репликация, което от своя страна улеснява хоризонталната мащабируемост. В този смисъл, Azure Kubernetes Service (AKS) е облачната PaaS услуга, която обгръща .NET Core и Docker приложения, като използва силата на Kubernetes, като същевременно се възползва от удобството и функциите на Azure.

Въз основа на разгледаните до тук технологии и инструменти, следва да представим архитектурна диаграма, която да съответства на концептуалните решения от втора глава и същевременно да се използват по-горе описаните облачни услуги.

 Фиг. 2.1. Архитектурна диаграма . (разработка на автора)

Важно да се отбележи, е че виртуализацията, контейнеризацията и наблюдението са основни практики, с които се занимава екипът по "Развитие" и "Операции" (DevOps). Основната характеристика се изразява в подкрепа на процеса на автоматизация и съблюдаване на всяка една стъпка от софтуерната разработка – от интеграция, тестване и пускане на пазара до инфраструктурен мениджмънт. В този смисъл непрекъсната интеграция и доставка (CI/CD) представляват систематичен подход за усъвършенстване на процесите на разработка и внедряване.



Фиг. 3.5. Схема на основен DevOps работен поток. Източник: Microsoft< https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/>, [23.10.2023]

Според експерти в областта, концептуалният работен поток, представен на фиг. 3.5, започва с редактиране на програмен код, който бива качен в хранилище за контрол на версиите. Това задейства процедурите от етапа „непрекъсната интеграция“ (CI), при които се компилира нововъведения програмен код, последван от тестове за валидиране на вече работещите функционалности. При успешна валидация се генерира артефакт или както беше описано по горе – изображение. Следва втората фаза на т.н. „непрекъснато внедряване“, което включва автоматизирани задачи за публикуване на новата версия, които улесняват разпространението на артефакта в желаната среда. Според нас, този рационализиран процес позволява автономни и постепенни актуализации. Всяка микроуслуга следва да премине през изолиран цикъл на изграждане, тестване и внедряване, което позволява ефективно управление на версиите.

В този смисъл, стратегии за внедряване, често срещани в литературата и практиката за осигуряване на актуализации и поддържане на стабилност в системата, са обобщени в табл. 3.4.

Таблица 3.4.  
 Обобщение на стратегии за внедряване, описани в теорията и често срещани в практиката  
(разработка на автора)

|  |  |
| --- | --- |
| **Стратегия за внедряване** | **Описание** |
| Синьо-зелено внедряване  (blue-green deployment) | Синьо-зеленото внедряване позволява едновременното изпълнение на две идентични производствени среди. „Синята“ представлява активната, докато „зелената“ обозначава новата версия. Идеята на този подход е да се изпробва нова версия в среда, подобна на производствена, без да се прекъсва активната услуга. |
| Постепенно внедряване  (rolling deployment) | При постепенното внедряване новата версия на приложението се актуализира поетапно, като се публикуват няколко контейнера, а не всички наведнъж. Докато определен брой услуги поддържат стара версия, при възникването на проблем, процесът на внедряване може да бъде спрян и проблемът лесно да бъде локализиран. |
| Внедряване на Canary  (Canary release) | При внедряване на Canary, промяна обхваща малка подгрупа от потребители, преди бъде приложена към цялата инфраструктура. Целта е да се тества малка част от трафика, преди да се разпространи към по-широката потребителска база. |

За да допълним темата за софтуерно внедряване, следва да преминем към моделите за производствено тестване. Подобно на стратегиите за внедряване, тестването гарантира, че софтуерът функционира според очакванията в производствена среда. Тези модели могат да помогнат за предотвратяване на софтуерни дефекти, подобряване на устойчивостта на системата и поддържане на качество и надеждност. Според техническата литература, съществуват няколко модела, като A/B тестването е един от тях. В контекста на базирана в облака система за управление на поръчки, A/B тестването дава възможност за вземане на решения, като позволява едновременното внедряване на различни версии на системни подобрения или нови функции за подгрупи от потребители, като по този начин позволява изготвянето на сравнителни оценки на ефективността. Тази процедура, показана на фиг. 3.6, позволява оптимизирани модификации чрез разчитане на емпирични данни.



Фиг. 3.6. Диаграмата илюстрираща внедряване на облачна услуга, използвайки А/Б тестване.. Източник: АБВ <https:// >, [23.08.2023]

Според случая на употреба, представен в диаграмата по горе, клиентите биват сегментирани в две отделни групи: група A и B. Трафикът на група A е насочен към оригиналните сървъри (инстанции 1, 2, 3 и 4) чрез балансьор на натоварването. От друга страна, Група B се насочва към отделен клъстер (5 и 6), който е снабден с нова функционалност. Този подход улеснява тестването чрез постепенното внедряване, като по този начин се минимизира риска.

В тази връзка, модела за производствено тестване Chaos Engineering представен на фиг. 3.7, дава възможност за умишлено въвеждане на дефекти в системата по време на работа, като същевременно се наблюдават реакциите. Чрез идентифициране на уязвимости в контролирана среда, екипите могат проактивно да измислят решения за подобряване на устойчивостта на системата.



Фиг. 3.7. Диаграмата илюстрираща Chaos Engineering. Източник: АБВ <https:// >, [23.08.2023]

Основавайки се на цялостна оценка на системните изисквания, детайлност в тестване и устойчивост към грешки, комбинацията от Canary Release и Chaos Engineering може да осигури балансиран подход за поддържане на стабилност, като същевременно да се подобрява облачно базираната системата за управление на поръчки. Чрез сегментиране на внедряването е възможно да се наблюдава въздействието на промените в системата в реално време, а идентифицирането на грешки следва да адресира уязвимостите на системата.