Софийски университет „Св. Кл. Охридски”

Факултет по математика и информатика

Магистърска програма
„Софтуерни технологии”

Предмет: Архитектури на софтуерни системи

Зимен семестър, 2024/2025 год.

Реферат

Архитектурни стилове в автономни и интелигентни софтуерни системи

Автор:

Радослав Стоянов Велков, фак. номер 7MI3400582

януари, 2025

София

Съдържание

1 Въведение

1.1 Представяне на областта

С развитието на технологиите, автономните и интелигентните софтуерни системи придобиват все по-голямо приложение в различни индустрии. Автономните системи са проектирани да изпълняват задачи без пряка човешка намеса, като взимат самостоятелни решения и адаптират поведението си според промените в околната среда. Интелигентните системи разчитат на изкуствен интелект, за да анализират данни, да вземат решения и да усъвършенстват действията си чрез обучение (трениране на модел) и адаптация. Тези технологии предоставят нови възможности за автоматизация и оптимизация, правейки ги ключови в основни области от нашето забързано и имащо нужда от всевъзможни автоматизации ежедневие.

Софтуерните архитектури играят основна роля за осигуряване на различни видове технологични, бизнес и архитектурни качества на дадена софтуерна система. В случая на автономните и интелигентните системи основни такива са: надеждност, производителност, устойчивост, скалируемост, използваемост, гъвкавост и адаптивност. Изборът на подходящ архитектурен стил за реализация определя как системата ще обработва информация, как ще комуникира с други модули и как ще адаптира поведението си в динамични среди.

1.2 Значимост на темата

Приложенията на автономните и интелигентните системи се разширяват бързо в множество различни индустрии и основополагащи сфери. В транспортния сектор, например, непрекъснато се върви към интегриране на превозни средства, използващи интелигентни технологии за разпознаване на обекти и навигация, за да осигурят максимално безопасно движение без човешка намеса в управлението. В здравеопазването, интелигентни системи помагат при диагностика и терапия, използвайки големи бази от данни и статистики, с цел да предоставят прецизни предписания, лечения, препоръки и т.н. Във финансите, автоматизираните системи за търговия и управление на риска разчитат на сложни алгортими и архитектури за вземане на решения в реално време, предвиждане на загуби. В сферата на образованието, интелигентните системи помагат за персонализирано обучение, адаптирайки учебните материали спрямо нуждите и напредъка на всеки ученик. Те използват алгоритми за анализ на резултатите и предоставят препоръки в реално време, което подобрява ефективността на обучението и регулира ангажираността на учащите. В производствената индустрия автономни роботи и „умни“ системи се използват за автоматизиране на производствените процеси, като включват предсказуема поддръжка, управление на инвентара и оптимизация на производствените линии. Чрез машинно самообучение, тези системи анализират работните условия и се адаптират към тях, като повишават ефективността и намаляват разходите. Такива примери демонстрират критичността на надеждността в тези типове системи.

1.3 Архитектурен стил

Архитектурният стил в софтуерното инженерство е начинът, по който се организира структурата на една система на високо ниво. Той определя как компонентите на системата са подредени, как взаимодействат помежду си и какви ограничения се налагат върху тези взаимодействия. Архитектурните стилове предоставят набор от понятия и правила, които насочват дизайна на системата, като включват видове компоненти (напр. модули, услуги) и връзките между тях (напр. съобщения, API). Примери за архитектурни стилове са: слоестата архитектура (Layered Architecture), клиент-сървър (Client-Server model), модел-изглед-контролер (Model-View-Controller), които са и предмет на реферата, и други популярни като: микроуслугите (Microservices) и ориентираната към събития архитектура (Event-Driven Architecture). Те служат като основа за изграждане на сложни софтуерни системи, като осигуряват ясно структурирана и разбираема организация. [1][13][14]

1.4 Мотив за избор на съответните архитектурни стилове

Изборът ми на архитектурните стилове Model-View-Controller (MVC), Client-Server и Layered Architecture се основава на тяхната доказана ефективност и практическа приложимост при изграждането на сложни и мащабни софтуерни системи, които изискват висока надеждност, гъвкавост и адаптивност. Реших да се спра на тези стилове, защото всеки от тях предлага предимства, които ги правят подходящи за видовете софтуерни системи, които са предмет на този реферат.

MVC дава възможността да организирам системата чрез ясно разделение на отговорностите, като дефинирам отделни роли за данните и бизнес логиката, потребителския интерфейс и управлението на взаимодействието. Това е особено полезно за интелигентни системи, които изискват динамичен интерфейс и обработка на сложни данни, например в области като здравеопазването или образованието. Client-Server архитектурата е ключова за системи с множество взаимодействия, защото осигурява централизация на данните и услугите, което позволява обработка в реално време и координация на множество клиенти, като структурата ѝ дава възможност за ефективна комуникация между клиентите и сървъра, което я прави незаменима за разпределени среди. Layered Architecture, от своя страна, осигурява структурираност чрез разделяне на функционалностите на различни слоеве, като презентационен слой, бизнес логика и слой за управление на данни. Този подход е особено подходящ за системи, които изискват управление на сложни зависимости, разширяемост и лесна интеграция с други системи.

Смятам, че комбинирането или индивидуалното използване на тези архитектурни стилове предлага оптимални решения за изграждането на стабилни, адаптивни и мащабируеми автономни и интелигентни системи, които могат ефективно да отговорят на динамичните изисквания на съвременния технологичен свят.

1.5 Текущо състояние на областта и проблеми

Както описахме, автономните и интелигентните системи се развиват значително през последното десетилетие и намират все по-широко приложение в различни сфери. Въпреки

този прогрес, областта е изправена пред редица предизвикателства, свързани с архитектурния дизайн и осигуряването на необходимите качествени показатели. Традиционните архитектурни стилове имат доказана ефективност при изграждане на класически софтуерни системи, но в контекста на автономни и интелигентни технологии те могат да срещнат ограничения.

Сред основните проблеми са:

- *Надеждност и устойчивост*: Този тип системи трябва да функционират без човешка намеса и да реагират адекватно на неочаквани промени. Дадени архитектурни стилове не винаги осигуряват необходимото ниво на отказоустойчивост и очаквано поведение в динамични условия.
- *Производителност и скалируемост*: Тези системи често обработват големи обеми данни и изискват висока производителност в реално време, което не винаги е лесно за постигане с традиционните архитектурни подходи, особено с такива предназначени за по-малки проекти като мащаб.
- *Гъвкавост и адаптивност*: За интелигентните системи е важно да могат да се самообучават и да адаптират поведението си спрямо промение в средата. Не всички стилове винаги позволяват лесна адаптация без значителни промени в кода.
- *Сигурност*: Динамичните условия и взаимодействието с външни системи повишават риска от атаки и злоупотреби, правейки сигурността на данните с висок приоритет сред тези технологии.

Такъв тип ограничения и, като цяло, адекватния анализ и правилния подход относно избор на архитектурен шаблон, който да се следва при реализация на системата, могат да доведат до противоречия и проблеми сред заинтересованите лица.

1.6 Принос към решението на проблемите и реализация

Настоящият реферат има за цел да изследва и анализира как архитектурните стилове Model-View-Controller, Client-Server и Layered могат да бъдат адаптирани и усъвършенствани, за да се справят с предизвикателствата на автономните и интелигентните системи. Това включва следните основни стъпки:

- *Анализ и адаптация*: Ще бъде извършен детайлен анализ на характеристиките на всеки от трите архитектурни стила и как съответният отговаря на нужните изисквания, изброени по-горе. Ще се предложат конкретни подобрения и адаптации за прилагането им в контекста на този вид софтуерни системи. Ще бъдат представени силните и слабите страни на всеки стил.

- *Класификация на приложимостта*: Ще се извърши класификация, която да очертае подходящите архитектурни стилове за различни типове автономни и интелигентни приложения, както и препоръки за оптимално използване на съответните стилове в различни случаи. Ще се използват конкретни примери от индустрията, които ще илюстрират използването на разглежданите архитектури в реални условия. Това ще предостави практически контекст и ще покаже възможностите и ограниченията на всеки стил.

- *Сравнение, включително с алтернативни архитектурни подходи*: За по-задълбочено разбиране ще бъде направено сравнение между стиловете, както помежду им, така и с някои алтернативни архитектури, използвани в областта на тези системи, като event-driven и microservices, с цел откриване на най-подходящите решения за специфични изисквания. [13][14]

- *Заключение и насоки за бъдещо развитие*: Ще бъде направен обобщен извод за приложимостта на MVC, Client-Server и Layered архитектурите в автономните и в интелигентните системи, давайки и насоки за бъдещи подобрения и надграждания.

2 Описание на засегнатите видове софтуерни системи

2.1 Интелигентни системи

Интелигентните софтуерни системи са върхът на съвременните технологични иновации, имплементирайки машинно самообучение чрез сложни алгоритми и статистика, за да анализират и интерпретират огромни обеми от данни. Те са не просто инструменти, а активни участници в процесите на вземане на решения, които се основават на адаптивност и самоподобрение във всеки един момент. Тези системи се захранват от т.нар. изкуствен интелект (artificial intelligence), който предоставя способността да се разпознават модели, да се правят прогнози и да се генерират отговори на основата на предходен опит. С времето интелигентните системи стават "по-умни", благодарение на обучението с данни в реално време. Колкото повече натрупани взаимодействия, било то с реален потребител или с всевъзможна информация, толкова повече се доближават до имитацията на истински (човешки) интелект, съответно персонализирайки препоръките си за дадената приложна област.

Интелигентните системи намират приложение в почти всяка сфера на живота. Примери за това могат да бъдат: здравеопазването, където могат да идентифицират болести чрез обработка на медицински изображения и да предлагат най-добрите терапии въз основа на пациентската история и вече изпитани методи на лечение; в транспорта, където са способни да управляват трафика оптимално, по този начин предотвратявайки задръствания и други неудобства; в индустрията; в образованието и като цяло във всяка една сфера.

Ключовата характеристика на интелигентните системи е тяхната способност да вземат решения, обикновено за пренебрежимо кратко време, тъй като те са проектирани да "мислят" и да реагират на ситуации, за които не са били изрично програмирани, което ги прави изключително ценни в сложни и динамични среди. Основни съставни техни компоненти на тази технология са: машинното самообучение (Machine Learning), обработката на естествен език (Natural Language Processing), роботиката, както и интеграцията в експертни система.

Въпреки впечатляващите им възможности, те имат и своите предизвикателства. Едно от тях е необходимостта от огромни обеми качествени данни, което може да е трудно постижимо в някои индустрии. Друго нещо, което в никакъв случай не е за подценяване са всевъзможните етични въпроси и казуси, което е една напълно различна тема. А както споменахме във

въведението, тези софтуери трябва да бъдат надеждни, с максимална производителност и гъвкави.

Интелигентните системи са не просто софтуер – те са визия за бъдещето, в което технологиите работят рамо до рамо с хората, за да създават по-ефективни, адаптивни и удобни решения. [2][3]

2.2 Автономни системи

Автономните системи са технологичен пробив, който променя представите ни за автоматизация и самостоятелна работа. Те са създадени с цел да извършват действия без пряка човешка намеса, като използват сензори, алгоритми и изчислителна мощ, за да възприемат и анализират обкръжението си с цел адекватна реакция. Основната им характеристика е тяхната способност да вземат решения в реално време, базирани на текущи условия и предварително зададени цели.

Автономните системи намират приложение в много и разнообразни сфери, като например:

- В транспорта, автономните превозни средства са пример за системи, които могат да анализират пътната обстановка, да предвиждат действията на участниците в движението и да вземат решения за безопасна и оптимална навигация.
- В роботиката, автономни роботи работят в труднодостъпни или опасни среди, като подводни експедиции, спасителни операции или дори космически мисии.
- В индустрията, автономните дронове инспектират инфраструктура, доставят стоки, изпълняват задачи с висока точност и без нуждата от човешка намеса.

Автономността изисква хармонична работа на няколко компонента в дадена последователност. Основния принцип при тези системи е че те разчитат на сензори (вкл. камери, радари, и др.), за да възприемат околната среда. След това чрез комплексни алгоритми (вкл. и такива с изкуствен интелект) анализират получената информация, като идентифицират обекти, определят рискове и изчисляват най-добрите потенциални действия. След дългия процес на анализ, се стига до използване на механизми за изпълнение, като двигатели, манипулатори или управляващи модули, за да реализират взетите решения.

Въпреки впечатляващия им напредък, автономните системи могат да се сблъскат предизвикателства от вида на: надеждност, що се отнася до осигуряването на стабилност при екстремни условия или непредвидени (т.е. рядко срещащи се) ситуации; етика и отговорност – и тук, логично, както при системите с ИИ, изникват редица етични въпроси като кой носи отговорност при грешка – системата, разработчикът или потребителят и т.н.; сигурност – при такъв тип системи винаги съществува потенциал за уязвимост към кибератаки, които могат да компрометират тяхната функционалност и желано поведение, в крайни случай нанасяйки поражения върху ползвателите им, които биха могли да бъдат дори фатални.

Автономните системи демонстрират как технологиите могат да се развият от инструменти в пълноправни „агенти“, способни да вземат самостоятелни решения. Те

променят бъдещето на човешките лични и професионални начинания, като създават възможности за безопасност, ефективност, оптималност и иновации на ново ниво. [4]

2.3 Автономни интелигентни системи

Автономните интелигентни системи представляват силна комбинация между способността за самостоятелно действие и използването на изкуствен интелект за вземане на информирани, адаптивни решения. Те комбинират ключовите характеристики на интелигентните системи – умението да се учат, анализират и подобряват своите действия, с автономността – способността да се действа независимо, без човешка намеса. Тази интеграция превръща автономните интелигентни системи в изключително мощни инструменти, които не само изпълняват задачи, но и се адаптират към променящи се условия и оптимизират действията си с течение на времето.

Такъв тип системи все повече намират реално приложение в множество области, трансформирайки както начина, по който се изпълняват задачи, така и изискванията към сигурност, ефективност и иновации. Такива примери са:

- Автономни автомобили с изкуствен интелект**:** Превозни средства анализират милиони данни в реално време – от сензори, камери, датчици, за да вземат решения за безопасно шофиране, като разпознават обекти, прогнозират движението на другите участници в трафика и адаптират поведението си спрямо пътните условия. Освен това тези системи се учат, като подобряват представянето си чрез обработка на огромни бази от данни, събрани от предишни ситуации.
- Медицински роботи**:** В здравеопазването автономните интелигентни системи съчетават прецизността на роботизираната хирургия с мощни аналитични алгоритми. Например, система може да извършва сложни хирургични процедури с минимална човешка намеса, като в същото време използва алгоритми за машинно обучение, за да оптимизира движенията си, намалявайки времето за операция и риска за пациента.
- Дронове за спешна помощ**:** В спасителни операции автономните интелигентни дронове се използват за търсене на оцелели след природни бедствия. Те са оборудвани с камери, термални сензори и алгоритми за изкуствен интелект, които позволяват да идентифицират хора в затрупани зони. Благодарение на автономните си възможности, тези дронове могат да навигират в труднодостъпни райони, а интелигентните им компоненти помагат за приоритизиране на зоните за претърсване въз основа на вероятността за откриване на оцелели.

Тези и още много модерни примери могат да бъдат дадени за приложението на тази комбинация от „интелигентност“ и автономност в една мащабна софтуерна система. [5]

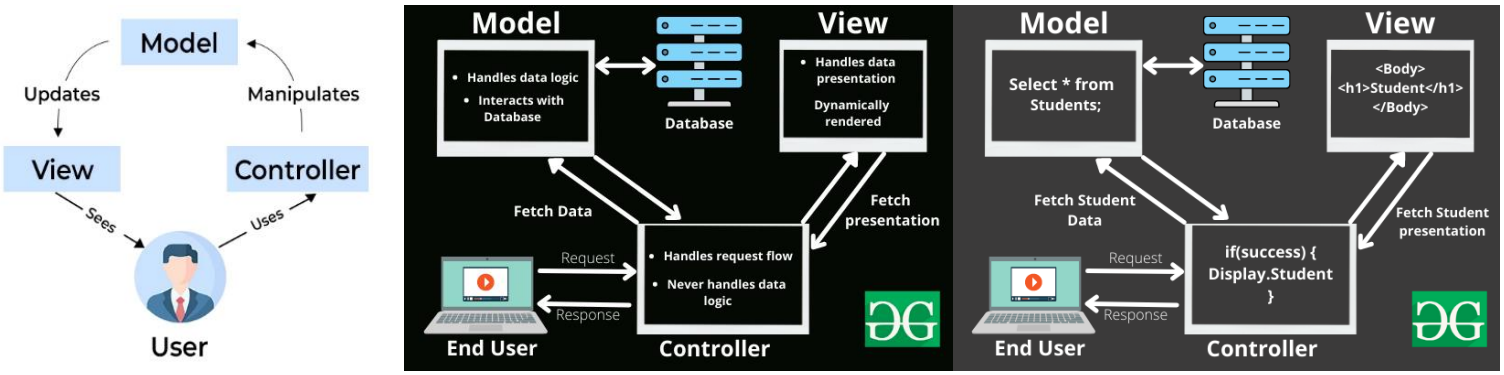
3 Анализ на архитектурните стилове и приложимост в такива системи

3.1 Model-View-Controller (MVC)

3.1.1 Основни характеристики

Model-View-Controller е един от най-ефективните архитектурни стилове за разделяне на отговорностите в софтуерните системи. Той осигурява ясно разграничение между логиката на приложението, начина, по който се представят данните, и взаимодействието с потребителя. Това го прави подходящ за сложни приложения, които изискват висока степен на модулност и адаптивност. Както името му подсказва е разделен на три съставни компонента:

- Model (Модел): Представлява ядрото на приложението, което управлява данните и бизнес логиката. Той обединява операции като съхранение, извличане и обработка на информация (обикновено от база данни), същевременно гарантирайки, че логиката остава независима от потребителския интерфейс. В интелигентните системи, моделът често включва алгоритми за машинно обучение или обработка на данни. Например, при автономен робот той би включвал модули за планиране на маршрути, разпознаване на обекти и прогнозиране на бъдещи състояния.
- View (Изглед)**:** Осигурява представяне на данните, подготвени от модела, във формат, удобен за потребителя. Този компонент е отговорен за визуализацията и за потребителския интерфейс, като обикновено включва текст, графики, графични интерфейси, аудио-визуални елементи и др. В автономни системи View-то може да се използва за показване на състоянията на системата, като например текущо местоположение, налични ресурси или аналитични данни за вземане на решения.
- Controller (Контролер): Действа като посредник между Model и View, като обработва входящите сигнали от потребителя или сензори и съответно задейства необходимите операции в модела. Той може да управлява взаимодействието между различни модули, като например вземането на решения въз основа на входни данни от околната среда. Например, в автомобил с автономно управление контролерът би обработвал входовете от камери и сензори и би предавал подходящи команди на модела за промяна на курса. [6][7]



Фигури 1/2/3. Илюстративни базови схеми на MVC (https://s2-labs.com/admin-tutorials/mvc-architecture/; https://www.geeksforgeeks.org/mvc-framework-introduction/)

3.1.2 Приложение

MVC архитектурата е изключително подходяща за разработка на автономни системи с висока степен на сложност и изискване за модулност. Чрез нея можем умело да управляваме сложността на цялостната софтуерна архитектура, поради факта, че структурираме системата в отделни компоненти, което значително улеснява поддръжката, разширяването и тестването. Например, ако автономно превозно средство се нуждае от нов алгоритъм за разпознаване на пътни знаци, той може да бъде добавен към Model-а без промени във View-то или Controller-а.

Друга важна характеристика, която се повлиява в положителна посока от MVC е гъвкавостта. Благодарение на ясното разделение на отговорностите, MVC позволява лесна адаптация към нови технологии. Например, ако потребителският интерфейс трябва да премине от уеб базиран към мобилен, промяната ще засегне само изгледът, докато модела и контролера остават непроменени. Също, автономните системи често изискват обработка на данни в реално време. MVC улеснява това, като позволява паралелна работа на компонентите. Например, докато контролерът обработва нов вход, модела може да подготвя данни за следващата итерация, а изгледът показва резултатите от предходната.

При интелигентните системи, използващи облачна инфраструктура, Model-View-Controller може да бъде разпределен между различни машини или устройства. Model може да бъде хостван в облака за тежки изчисления, докато View и Controller остават на локалното устройство, осигурявайки бърза реакция на входни данни. По този начин се предоставя една модулност на компонентите (елементите) в разпределени среди.

Като цяло можем да обобщим, че MVC архитектурният стил е особено подходящ за автономни и интелигентни системи, поради налагащата се ясно структурирана архитектура и модуларност, както и възможност за лесно адаптиране към динамични условия. [6][7]

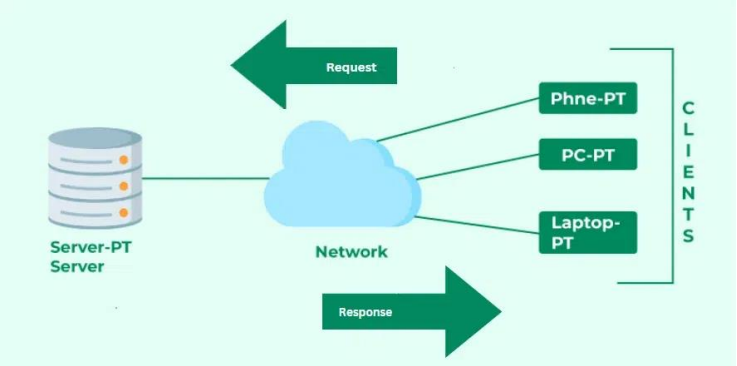
3.2 Client-Server

3.2.1 Основни характеристики

Клиент-сървър е един от най-широко използваните архитектурни стилове, характеризиращ се с ясно разделение на ролите между клиентите, които инициират заявки, и сървърите, които обработват тези заявки. Тази архитектура е особено подходяща за системи с децентрализирана обработка на данни и множество потребители или устройства.

Т.нар. „клиент“ е компонент, който изпраща заявки до „сървъра“ и предоставя интерфейс за взаимодействие с крайния потребител. Клиентите обикновено се изпълняват на крайни устройства (напр. лаптопи, смартфони, IoT устройства) и често съдържат само графичен потребителския интерфейс, заедно с малка част от логиката. При интелигентни системи, клиентите могат да бъдат автономни устройства (например дронове или роботи), които изпращат данни за състояние или заявки за изчисления към сървъра, който, от своя страна, е централизираният компонент, обработващ заявките от клиентите, съхраняващ данните и предоставящ необходимите услуги. Сървърите често разполагат с по-мощни изчислителни ресурси и обикновено поддържат сложни алгоритми за обработка на всевъзможни набори от данни. Практика е да включват компоненти за анализ на големи данни, машинно обучение,

вземане на решения (особено в системите с ИИ). Например, сървърът в система за интелигентно управление на трафика би обработвал входящите данни от множество сензори и би предоставял оптимални маршрути за автомобили. [8][9]



Фигура 4. Илюстративна базова схема на Client-Server (https://www.geeksforgeeks.org/client-server-model/)

3.2.2 Приложение

Клиент-сървър архитектурата позволява централизирано управление на данните и услугите, което улеснява поддръжката и осигурява мащабируемост. Това е особено полезно в системи с множество клиенти, като мрежи от IoT устройства или свързани автономни автомобили. В много от този вид системи, клиентите са ограничени по отношение на изчислителната мощност. Не е тяхна отговорност самостойно да извършват тежка логическа работа. Тази архитектура позволяв прехвърлянето на сложните задачи към сървъра. Например, дрон с ограничен хардуер може да изпраща сензорни данни до облачен сървър, който обработва информацията чрез алгоритми за изкуствен интелект и връща команди за управление на хардуерните модули.

Клиентът и сървърът, от друга страна, могат да бъдат проектирани така, че да изпълняват различни части от общата функционалност, което ще е с цел да се увеличи ефективността на системата. В допълнение на това, сървърът може да бъде разположен в облака, което осигурява достъп до значителни ресурси за съхранение и обработка на големи данни. Това е важно за автономни системи, които изискват обработка на големи обеми информация. Приложен пример тук може да се даде със система за интелигентно здравеопазване, в която IoT устройствата (клиентите) събират биометрични данни от пациентите и ги изпращат към облачен сървър, който извършва анализ и предоставя персонализирани медицински препоръки.

Друг важен аспект е фактът, че централизираната структура улеснява откриването и разрешаването на грешки. Ако даден клиент изпита проблем, сървърът може да поеме част от функциите му временно или да преразпредели задачите към други клиенти. Колкото повече нараства мащаба на архитектурата/системата, толкова повече трябва да се взима предвид това свойство.

Client-server архитектурата може да бъде от полза за разработващите при следните видове софтуерни системи:

- Управление на трафик: в интелигентни транспортни системи „клиентите“ могат да бъдат превозни средства, които изпращат данни за своето местоположение, скорост и състояние до едно централен сървър. Той ще е отговорен да обработва динамично пристигащата информация и да предоставя оптимални маршрути , предупреждения за инциденти, оптимална скорост и др.
- Разпределени автономни роботи: роботи в производствена среда могат да действат като клиенти, които комуникират със сървър, управляващ ресурсите и координиращ действията им, за да оптимизира ефективността на производството.
- Смарт градове: IoT устройства в градската инфраструктура (като сензори за осветление или паркинги) ще изпращат данни към централен сървър, който преценява как да разпредели използването на ресурси, например, като регулира осветлението или насочва шофьорите към свободни паркоместа.

Като цяло, „Клиент-сървър“ е ключов архитектурен стил за автономни и интелигентни системи, осигуряващ ефективност, мащабируемост и възможност за централизирано управление. Особено подходящ за системи с голям брой взаимодействащи равнопоставени устройства, които разчитат на обработка на данни в реално време. [8][9]

3.3 Layered/Multitier (Многослойна архитектура)

3.3.1 Основни характеристики

Многослойната архитектура е структура, която разделя софтуерната система на отделни слоеве (или също наричани нива), всеки от които има ясно дефинирана отговорност. Този архитектурен стил улеснява организацията, поддръжката и развитието на сложни системи, правейки ги многопластови. Всеки слой предоставя услуги на слоя над него и използва услуги от слоя под него. Чрез разделянето на системата на слоеве, тя става по-лесна за поддръжка, разширение и тестване.

Типичните слоеве включват:

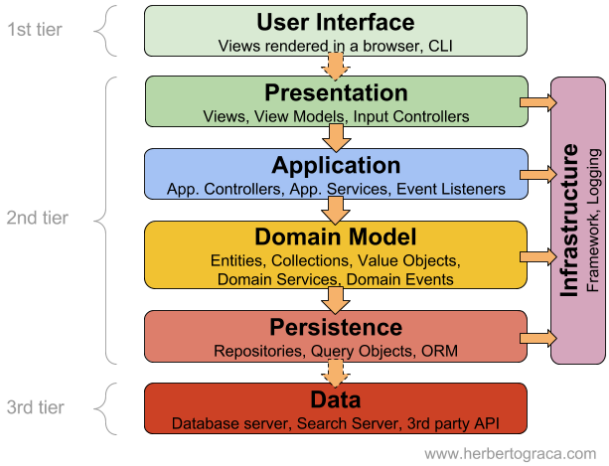
- Презентационен слой (Presentation Layer): Този слой е отговорен за взаимодействието между потребителите и системата. Той е „лицето“ на приложението, което предоставя интерфейси, чрез които потребителите въвеждат данни, изпращат заявки и получават обратна връзка. Основните му отговорности са да създава интуитивен потребителски интерфейс, подсигурявайки добро потребителско преживяване, да валидира въведените от потребителя данни преди те да бъдат обработени от бизнес слоя, както и да превръща отговорите от по-долните слоеве в желан вид.

- Бизнес слой (Business Layer): Съдържа бизнес логиката и управлява основните правила и процеси (функционалности), които системата трябва да изпълнява, за да отговори на изискванията на потребителите и бизнес нуждите. Обработва заявки от презентационния слой, управлява данните и последователността на операциите и контролира цялостния работен процес.

Този слой предразполага към значителна комплексност според вида на системата и може да бъде разделен, като част от неговите функционалности могат да бъдат поети от „приложен слой/слой за услуги“ (Application/Service layer). Така той ще се явява посредник между презентационния и бизнес слоя, съдържайки логика за координация.

- Данни (Data Layer): Този слой се занимава с достъпа, съхранението и управлението на данните, използвани от системата. Той комуникира с базите данни, хранилищата или външните системи, за да осигури необходимите данни за горните слоеве. Отговорен е и да поддържа целостта и сигурността на данните.

- Интеграционен слой (Integration Layer): Не винаги бива включван. Той действа като посредник, който осигурява комуникацията между различни модули на системата или между нея и други външни услуги. Може да включва API, съобщителни системи, протоколи за интеграция. Иначе казано, координира обмяната на информация между системите и поддържа съвместимост между различни технологии и формати. [10][11][12]



Фигура 5. Илюстративна базова схема на Multitier architecture (https://herbertograca.com/2017/08/03/layered-architecture/)

3.3.2 Приложение

Многослойната архитектура предоставя ясно разделение на отговорностите, което улеснява декомпозирането на отделни функции и модули. Презентационният слой, например, може да служи като интерфейс за наблюдение и управление на автономна система. Бизнес слоят е мястото, където се реализират алгоритмите за вземане на решения или машинно обучение, а данните могат да бъдат съхранявани в облачна инфраструктура или локални бази данни. Това разделение подобрява организационната структура на системата и прави развитието ѝ по-ефективно.

Едно от основните предимства на многослойната архитектура е модулността ѝ и лесната ѝ поддръжка. Чрез разделяне на системата на слоеве сложността при промени значително се намалява. Например, ако е необходима оптимизация на бизнес логиката, тя

може да бъде извършена, без да се налага модификация на другите слоеве. Освен това, многослойната архитектура осигурява гъвкавост при внедряване. Всеки слой може да бъде разпределен на различни физически или виртуални машини, което улеснява както разширяемостта и скалируемостта, така и адаптацията към различни среди и като цяло платформената независимост. Нови слоеве или функционалности могат да бъдат добавени към системата, без това да налага значителни промени в съществуващите компоненти. Например, в интелигентен дом може лесно да бъде добавен нов модул за управление на енергията, без да се засягат слоевете, отговорни за сигурността или осветлението. Така, например, презентационният слой може да бъде реализиран като уеб или мобилно приложение, докато бизнес и интеграционните слоеве могат да функционират в облака. Друго съществено предимство е възможността за интеграция с други системи. Интеграционният слой играе ключова роля в това отношение, като улеснява взаимодействието между автономните системи и външни услуги. Това е изключително важно за интелигентни системи, които трябва да комуникират с различни модули, устройства или външни API. Тази структура прави многослойната архитектура подходящ избор за изграждането на сложни автономни и интелигентни системи.

Следните реални примери за приложимост на n-tier архитектурата, в обсега на автономните и интелигентните системи, са визуализирани под формата на таблица с редове – същността на примерната система и колони – подходящите слоеве от софтуерната ѝ архитектура: [10][11][12]

Приложение/ Софт. система	*Презентационен слой*	*Бизнес слой/ Приложен слой*	*Данни*	*Интеграционен слой*
Автономни превозни средства	Интерфейс за водача или център за управление	Алгоритми за автономно управление, разпознаване на обекти и вземане на решения	Локална или облачна база данни за карти, сензорна информация и предходни данни за управление	Комуникация с други превозни средства, пътна инфраструктура или облачни услуги
Интелигентни фабрики	Потребителски интерфейси за оператори и инженери	Логика за оптимизация на производствения процес, прогнозна поддръжка и разпределение на задачи	Интеграция с бази данни за инвентаризация, производствен капацитет и сензорни данни от машини	Връзка със системи за „Планиране на ресурсите в предприятието“ или IoT платформи
Системи за интелигентно здравеопазване	Мобилни приложения или уеб интерфейси за пациенти и лекари	Логика за диагностика, анализ на данни от апаратура или планиране на терапии	Централизирани здравни досиета и данни за предишна история на пациента	Комуникация с други здравни институции и национални регистри.

4 Обобщение – начин за избор на подходящ архитектурен стил

Изборът на архитектурен стил за една софтуерна система зависи от спецификите на задачата, която трябва да се реши, и от изискванията към структурата и функционирането на системата. Следвайки концепцията if-elseif-else или пък тази на блок-схемите, можем да определим кога е най-подходящо да изберем Model-View-Controller (MVC), Client-Server или Layered архитектура, като разгледаме ключовите характеристики и нужди на системата.

Ще опишем методът, използвайки следния формат *„[стил]*: *[характеристики]“* – избираме дадения [стил], ако системата изисква следните [характеристики].

- Model-View-Controller:
    - ясно разделение на отговорностите;
    - лесно обновяване на отделни компоненти;
    - интерактивен (динамичен) интерфейс за потребителя;
    - потенциал за интеграция на нови технологии.
- Client-Server:
    - работа в разпределена среда;
    - множество взаимодействащи устройства/единици;
    - централизирана обработка на данни;
    - управление на голям обем данни, включително паралелни.
- Layered:
    - повишена комплексност;
    - изразена модулност;
    - разделяне на различни функционални слоеве;
    - висока степен на разширяемост;
    - мащабируеми системи;
    - лесно проследяване и управление на зависимостите между различни части на системата.

5 Заключение

В този реферат разгледахме основните архитектурни стилове – Model-View-Controller (MVC), Client-Server и Layered – и тяхната приложимост в интелигентни, автономни и комбинацията от двете системи. Изследването подчерта значението на архитектурните решения за проектирането на такива системи, които не само трябва да изпълняват сложни задачи, но и да демонстрират висока надеждност, адаптивност и ефективност.

Първоначално направихме анализ на видовете системи, обект на текущия реферат, като подчертахме техните ключови характеристики и примери за реализация. Интелигентните системи, захранвани от изкуствен интелект, са способни да анализират данни, да се учат и да вземат решения, докато автономните системи разчитат на самостоятелни действия, базирани на събиране и обработка на данни в реално време. Обединявайки тези качества, автономните интелигентни системи съчетават самостоятелност и имитация на човешки интелект, което ги прави мощни инструменти в различни индустрии.

При анализа на архитектурните стилове се установява, че:

1. MVC предлага модулност и ясно разделение на отговорностите, което го прави подходящ за приложения, изискващи структурираност, например в интелигентни образователни системи или системи за здравеопазване.

2. Client-Server осигурява централизация и мащабируемост, като е идеален за разпределени системи, каквито са мрежите от IoT устройства или автономните превозни средства.

3. Layered позволява структурираност и разширяемост, което го прави подходящ за сложни приложения, като автономни фабрики и интегрирани здравни системи.

Всеки архитектурен стил демонстрира специфични предимства в зависимост от изискванията на системата, като тяхното комбиниране или адаптиране може да доведе до оптимални резултати. Примерите, включени в реферата, показват реалното приложение на тези архитектури, подчертавайки тяхната важност за развитието на съвременните интелигентни и автономни технологии.

Основните изводи са, че ясното разделение на отговорностите при MVC и слоест стил улеснява адаптирането към променящи се изисквания, като същевременно запазва устойчивостта на системата, в допълнение, клиент-сървърната архитектура е изключително ефективна за системи с множество взаимодействия и обработка на данни в реално време, а пък многослойната и клиент-сървър осигуряват лесна интеграция с външни системи, което е от съществено значение за взаимодействието между автономни устройства и глобални мрежи.

5.1 Идеи за бъдещо развитие

1. Хибридни архитектури: Комбинирането на MVC, Client-Server и Layered може да създаде нови възможности за оптимизация, особено при сложни автономни системи.

2. Проучване на нови архитектурни стилове: Стилове като Event-Driven и Microservices могат да предложат алтернативи за приложения, изискващи динамичност и разпределеност.

3. Етика и сигурност: Тъй като автономните и интелигентните системи стават все по-широко разпространени, бъдещите изследвания трябва да се съсредоточат върху интегрирането на етични принципи и гарантирането на сигурността на данните.

6 Използвана литература

[1] *Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media. 2020.;*
Design Patterns: Elements of Reusable Object-Oriented Software.;
Patterns of Enterprise Application Architecture.
(чрез https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns)

[2] *https://www.unr.edu/cse/undergraduates/prospective-students/what-are-intelligent-systems*

[3] *https://www.geeksforgeeks.org/intelligent-systems-in-ai/*

[4] *https://www.sciencedirect.com/topics/computer-science/autonomous-system*

[5] *https://www.datasciencecentral.com/autonomous-intelligent-systems/*

[6] *https://www.geeksforgeeks.org/mvc-design-pattern/*

[7] *Burbeck, Steve (1992) Applications Programming in Smalltalk-80:How to use Model–View–Controller (MVC);*
LaLonde, Wilf R.; Pugh, John R. (1991). Inside Smalltalk. U.S.A.: Prentice-Hall Inc. p. 8. ISBN 0-13-467309-3. The model can be any object without restriction.;
WebObjects System Overview (PDF). Cupertino, CA: Apple Computer, Inc. May 2001. p. 28. In WebObjects, a model establishes and maintains a correspondence between an enterprise object class and data stored in a relational database.;
"Active Record Basics". Rails Guides. Retrieved October 27, 2022. This will create a Product model, mapped to a products table at the database.;
LaLonde, Wilf R.; Pugh, John R. (1991). Inside Smalltalk. U.S.A.: Prentice-Hall Inc. p. 8. ISBN 0-13-467309-3. The view is responsible for providing a visual representation of the object.;
LaLonde, Wilf R.; Pugh, John R. (1991). Inside Smalltalk. U.S.A.: Prentice-Hall Inc. p. 8. ISBN 0-13-467309-3. [MVC] permits views to be used as parts for assembly into larger units; new kinds of views can be constructed using existing views as subviews.;
LaLonde, Wilf R.; Pugh, John R. (1991). Inside Smalltalk. U.S.A.: Prentice-Hall Inc. p. 9. ISBN 0-13-467309-3. ...the view knows explicitly about the model and the controller.;
Simple Example of MVC (Model–View–Controller) Architectural Pattern for Abstraction;
Leff, Avraham; Rayfield, James T. (September 2001). Web-Application Development Using the Model/View/Controller Design Pattern. IEEE Enterprise Distributed Object Computing Conference. pp. 118–127.;
Buschmann, Frank (1996) Pattern-Oriented Software Architecture.;
Gamma, Erich et al. (1994) Design Patterns;
Moore, Dana et al. (2007) Professional Rich Internet Applications: Ajax and Beyond: "Since the origin of MVC, there have been many interpretations of the pattern. The concept has been adapted and applied in very different ways to a wide variety of systems and architectures.".
(чрез https://en.wikipedia.org/wiki/Model–view–controller)

[8] *https://www.geeksforgeeks.org/client-server-model/*

[9] *"Distributed Application Architecture" (PDF). Sun Microsystem. Archived from the original (PDF) on 6 April 2011. Retrieved 2009-06-16.;*
*Benatallah, B.; Casati, F.; Toumani, F. (2004). "Web service conversation modeling: A cornerstone for e-business automation". IEEE Internet Computing. **8**: 46–54.;*

*Dustdar, S.; Schreiner, W. (2005). "A survey on web services composition" (PDF). International Journal of Web and Grid Services. **1**: 1. CiteSeerX 10.1.1.139.4827;*
"What are the differences between server-side and client-side programming?". softwareengineering.stackexchange.com. Retrieved 2016-12-13.;
Nieh, Jason; Yang, S. Jae; Novik, Naomi (2000). "A Comparison of Thin-Client Computing Architectures". Academic Commons. doi:10.7916/D8Z329VF. Retrieved 28 November 2018.;
d'Amore, M. J.; Oberst, D. J. (1983). "Microcomputers and mainframes". Proceedings of the 11th annual ACM SIGUCCS conference on User services - SIGUCCS '83. p. 7;
*Tolia, Niraj; Andersen, David G.; Satyanarayanan, M. (March 2006). "Quantifying Interactive User Experience on Thin Clients" (PDF). Computer. **39** (3). IEEE Computer Society: 46–52.;*
Otey, Michael (22 March 2011). "Is the Cloud Really Just the Return of Mainframe Computing?". SQL Server Pro. Penton Media. Archived from the original on 3 December 2013. Retrieved 1 December 2013.;
*Barros, A. P.; Dumas, M. (2006). "The Rise of Web Service Ecosystems". IT Professional. **8** (5): 31.*
(чрез https://en.wikipedia.org/wiki/Client–server_model)

[10] https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

[11] https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier

[12]Richards, Mark (2020). Fundamentals of Software Architecture: An Engineering Approach (1st ed.). O'Reilly Media.;
Richards, Mark (2022). Software Architecture Patterns. O'Reilly Media, Inc.;
Deployment Patterns (Microsoft Enterprise Architecture, Patterns, and Practices);
Fowler, Martin "Patterns of Enterprise Application Architecture" (2002). Addison Wesley.;
Vicente, Alfonso; Etcheverry, Lorena; Sabiguero, Ariel (2021). "An RDBMS-only architecture for web applications". 2021 XLVII Latin American Computing Conference (CLEI). pp. 1–9.;
Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael (1996-08). Pattern-Oriented Software Architecture, Volume 1, A System of Patterns. Wiley, August 1996. ISBN 978-0-471-95869-7. Retrieved from http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471958697.html;
Martin Fowler's Service Layer;
Martin Fowler explains that Service Layer is the same as Application Layer;
Domain-Driven Design, the Book pp. 68-74. Retrieved from http://www.domaindrivendesign.org/books#DDD;
Richards, Mark (March 3, 2020). Fundamentals of Software Architecture: An Engineering Approach (1st ed.). O'Reilly Media.;
Eckerson, Wayne W. "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." Open Information Systems 10, 1 (January 1995): 3(20).
(чрез https://en.wikipedia.org/wiki/Multitier_architecture)

[13]Richards, Mark. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media. (чрез https://en.wikipedia.org/wiki/Event-driven_architecture)

[14] Fowler, Martin (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional. (чрез https://en.wikipedia.org/wiki/Microservices)