# Архитектура

# Контролер

Контролера представлява връзката помежду вътрешната modbus мрежа на клиента и централизираната сървърна система.

## Хардуер

Хардуера предвиден за прототип е микро контролер способен да подържа Linux базирана операционна система, да има сериен порт от тип usb, да има Ethernet порт и по възможност да има html out порт. Тези характеристики биват покрити от множествено микро контролери. Проучени микро контролери подходя за целта са:

Orange Pi Zero2:

Orange Pi Zero2 е малък едноплатков компютър, създаден от компанията Shenzhen Xunlong Software CO., Limited. Този компактен и достъпен компютър е предназначен за различни DIY проекти, разработка на вградени системи и IoT (Интернет на нещата) приложения.

**Основни характеристики на Orange Pi Zero2 включват:**

**Процесор и производителност**: Orange Pi Zero2 е оборудван с Allwinner H616 SoC, който включва четириядрен ARM Cortex-A53 процесор. Този процесор предлага добра производителност за широк кръг от приложения.

**Графика**: Вградената графика е Mali G31 MP2 GPU, която поддържа 3D графика и видео ускорение, позволявайки възпроизвеждане на висококачествено видео и графични приложения.

**Памет**: Устройството разполага с 512MB или 1GB DDR3 RAM, в зависимост от конкретния модел, което осигурява достатъчно памет за повечето леки до средно тежки задачи.

**Съхранение и разширение**: Orange Pi Zero2 разполага с microSD слот за разширяемо съхранение, което позволява инсталирането на операционни системи и приложения.

**Свързаност**:

**Wi-Fi и Bluetooth**: Вградена поддръжка за 2.4GHz Wi-Fi и Bluetooth 4.2, което улеснява безжичната свързаност и комуникация. **Ethernet**: 100Mbps Ethernet порт за стабилна кабелна мрежова връзка.

**Портове и интерфейси**:

**HDMI**: Пълноразмерен HDMI порт за свързване към дисплеи и телевизори.

**USB**: Различни USB портове за свързване на периферни устройства като клавиатури, мишки и други.

**GPIO**: General Purpose Input/Output пинове за свързване на сензори и други компоненти, което го прави идеален за хоби и професионални електронни проекти.

**Операционни системи**: Поддържа различни операционни системи като Android, Ubuntu и Debian, предоставяйки гъвкавост и лесна интеграция в различни среди.

Със своите компактни размери и мощни функции, Orange Pi Zero2 е подходящ както за начинаещи, така и за напреднали потребители, търсещи надеждно и ефективно решение за своите проекти и приложения.



Фиг. Изображение показващо Техническите характеристики на Orange Pi Zero2 както и всички свободни пинове и портове.

Radxa zero 3e:

Radxa Zero 3e е миниатюрен компютър, проектиран от компанията Radxa. Той е част от серията Radxa Zero и е известен със своята компактност и мощност, съчетавайки малки размери с отлични технически характеристики. Устройството е оборудвано с мощен процесор и достатъчно оперативна памет, което го прави подходящо за различни приложения, включително ембедед системи, Интернет на нещата (IoT) проекти и мултимедийни задачи.

Radxa Zero 3e разполага с множество интерфейси и портове, които позволяват свързване към различни периферни устройства и модули. Това включва HDMI изход за видео, USB портове за свързване на периферия и microSD слот за разширяване на паметта. Устройството поддържа различни операционни системи, което го прави гъвкаво и лесно за интегриране в различни проекти.

Със своята комбинация от мощност, компактност и гъвкавост, Radxa Zero 3e е привлекателен избор за разработчици и ентусиасти, които търсят ефективно решение за своите технологични нужди.



Фиг. Изображение представящо Radxa zero 3e

ROCK Pi S

ROCK Pi S е малък, но мощен едноплатков компютър, който използвам за различни проекти. Този миниатюрен компютър е проектиран от Radxa и е идеален за IoT (Интернет на нещата) приложения, както и за вградени системи.

Ето основните му характеристики, които ме впечатляват:

**Процесор и производителност**: ROCK Pi S разполага с Rockchip RK3308 четириядрен ARM Cortex-A35 процесор, който осигурява достатъчно мощност за различни задачи, въпреки малките си размери.

**Памет и съхранение**: Има опции с 256MB или 512MB RAM, което е достатъчно за леки приложения и IoT задачи. За съхранение използва microSD карта и може да поддържа и NAND флаш памет.

**Свързаност**:

**Wi-Fi и Bluetooth**: Вградените 2.4GHz Wi-Fi и Bluetooth 4.2 ме улесняват в безжичната комуникация и свързване с други устройства.

**Ethernet**: 10/100Mbps Ethernet порт предоставя надеждна кабелна мрежова връзка.

**Портове и интерфейси**:

**USB**: Има няколко USB порта, които използвам за свързване на периферни устройства.

**GPIO**: General Purpose Input/Output пиновете ми позволяват да свързвам различни сензори и модули, което го прави идеален за проекти с вградени системи.

**Размери и консумация на енергия**: ROCK Pi S е изключително компактен и енергийно ефективен, което го прави подходящ за проекти, където пространството и консумацията на енергия са критични.

**Операционни системи**: Поддържа различни операционни системи като Debian и Ubuntu, което ми дава гъвкавост при разработването и интегрирането на различни приложения.

Използвайки ROCK Pi S, успявам да реализирам различни идеи и проекти, независимо дали става въпрос за домашна автоматизация, IoT устройства или вградени системи. Този малък компютър е мощен инструмент в моите ръце.



Фиг. Изображение на едноплатковия компютър ROCK Pi S

За целта на проекта е решено да се вземе модела Orange pi zero 2

# Контрол на версиите

Контролът на версиите е система, която позволява проследяване на промените в даден проект с течение на времето. Той записва всяка модификация, направена върху файловете, като съхранява отделни версии на проекта. Това дава възможност на разработчика да се връща към предишни версии, да сравнява различни ревизии и да разбира кой е направил определени промени и защо. При работа в екип контролът на версиите позволява множество хора да работят върху един и същ проект едновременно, като предотвратява конфликтите, свързани с редактирането на един и същ код или файл.

## Технология за контрол на версиите

Git е разпределена система за контрол на версиите, която позволява на разработчиците да проследяват промените в техните проекти и да работят заедно по код. Тя съхранява пълна история на всяко изменение, като позволява лесно връщане към предишни версии на проекта. Git работи локално на компютъра на всеки потребител, което означава, че всеки разработчик има пълно копие на целия проект и неговата история.

Една от основните характеристики на Git е възможността за създаване на клони, което дава на разработчиците свободата да експериментират и работят върху нови функции, без да променят основния код. След като тези промени бъдат завършени и тествани, те могат да бъдат обединени с основния код. Git също така е много ефективен при управление на конфликтите, които възникват, когато няколко души работят едновременно върху един и същ проект.

Git се използва в днешно време, защото предоставя ефективен начин за управление на проекти и сътрудничество между разработчиците. Той позволява на екипите да проследяват промените в кода, да съхраняват историята на проекта и да работят паралелно без конфликти. Разработчиците ценят Git за неговата гъвкавост, тъй като могат да създават клони и да експериментират с нови функции, преди да ги обединят с основния код.

Git също така улеснява работата с отворен код, като позволява на множество хора да допринасят за един проект, независимо от тяхното местоположение. Системата за контрол на версиите осигурява стабилност и безопасност, тъй като позволява лесно възстановяване на предишни версии в случай на грешки. В допълнение, Git интегрира много инструменти и платформи, което го прави важен компонент в съвременния софтуерен процес на разработка.

## Конкуренцията за хостинг на Git

Конкуренцията за хостинг на Git включва платформи като GitLab, Bitbucket и други подобни услуги, които предоставят хранилище за код и инструменти за управление на проекти. Всяка от тези платформи предлага сходни функции, като контрол на версиите, интеграция с CI/CD и възможности за сътрудничество. GitLab, например, се използва заради своите мощни DevOps инструменти и опции за самостоятелен хостинг, докато Bitbucket се интегрира добре с други продукти на Atlassian, като Jira.

GitHub обаче се предпочита заради широкото му разпространение и голямата потребителска база. Въпреки че конкурентите предлагат специфични функции, GitHub остава водеща платформа за хостинг на Git проекти поради своята популярност, лесен интерфейс и силна интеграция с екосистемата на разработчиците.

## Платформа за хостинг на git

GitHub е онлайн платформа за хостване на софтуерни проекти и сътрудничество, базирана на системата за контрол на версиите Git. Тя позволява на разработчиците да качват своя код, да го споделят с други, както и да работят заедно по проекти, независимо от тяхното местоположение. GitHub предоставя функции като управление на версиите, проследяване на проблеми (issues), преглед на кода (code review), и създаване на клонинги (forks) на проекти. Той също така улеснява процеса на интеграция и доставка на софтуер (CI/CD), като е широко използван както от индивидуални разработчици, така и от компании по целия свят. В главата за автоматизация е разгледано как е приложени GitHub actions за проекта.

GitHub също така предлага възможност за създаване на хранилища (repositories), където кодът се съхранява и организира. Всяко хранилище съдържа пълна история на промените, направени върху проекта, което позволява на разработчиците да проследяват всяка версия и лесно да се връщат към предишни състояния на кода при нужда. Платформата поддържа концепцията за "pull requests", които позволяват на потребителите да предложат промени в кода на даден проект, а екипът на проекта може да прегледа и обсъди тези промени преди тяхното одобрение.

GitHub също така има социални аспекти – разработчиците могат да следват други потребители, да споделят проекти публично и да откриват нови проекти чрез търсене на хранилища или потребители с подобни интереси. Платформата е интегрирана с множество инструменти за разработка и автоматизация, което я прави мощен инструмент за управление на проекти и колаборация в света на софтуерното инженерство.

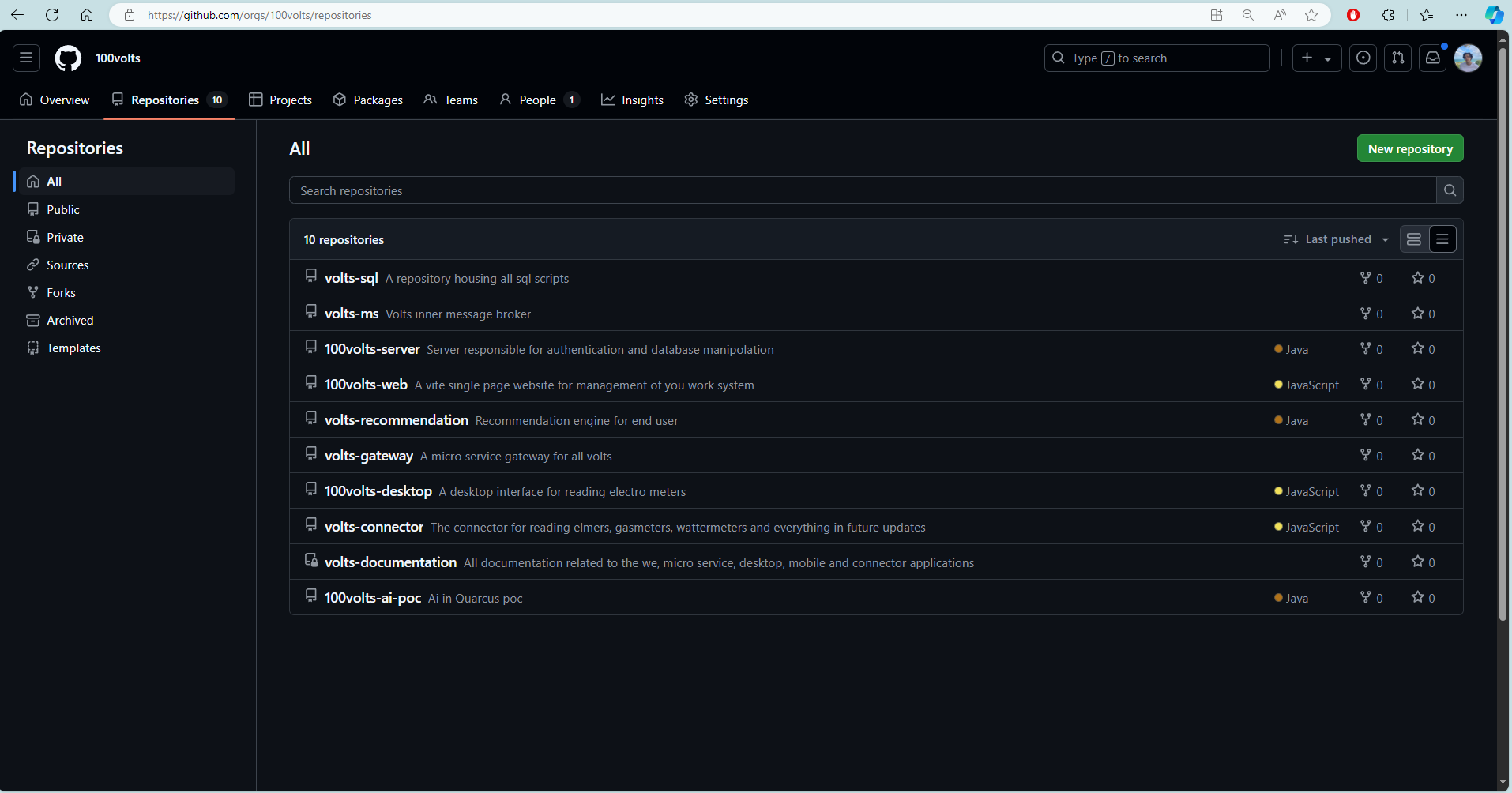
## Организация на репозиториите

Репозиторията представлява хранилище за код и свързаните с него файлове в рамките на проект. Тя съхранява цялата история на промените, направени върху проекта, включително различни версии, добавяния и редакции. Разработчикът използва репозиторията, за да организира файловете и да проследява работата си, като лесно може да преглежда предишни версии и да създава нови клонове за експерименти. Репозиторията позволява съвместна работа с други разработчици, като предоставя начин за споделяне и съхраняване на проекта на едно централно място, обикновено на платформа като GitHub.

GitHub организацията представлява група от потребители, която споделя и управлява проекти на едно централно място. Тя позволява на различни членове да работят заедно, като предоставя контрол върху достъпа и разрешенията за различни хранилища. Чрез организацията могат да се създават и управляват екипи с различни нива на права върху проектите, което улеснява координацията и сътрудничеството в екипа. Организацията също така осигурява възможност за управление на множество проекти едновременно, като централизира ресурсите и улеснява работния процес. За целите на проекта е работено от един човек но много рядко един програмен продукт се разработва само от един човек. Затова е избран стандарта за организиране на рапозитории чрез организация.

GitHub организацията се конкурира с платформи като GitLab и Bitbucket, които също предлагат хостинг на код и инструменти за сътрудничество. Въпреки че тези конкуренти предоставят сходни функции като управление на хранилища и интеграция с CI/CD, GitHub организацията се избира заради широката си популярност и голямата общност от разработчици. Тя има добре изградена екосистема и предлага отлична интеграция с инструменти като GitHub Actions за автоматизация.

GitHub организацията също така се предпочита заради лесния си за използване интерфейс и стабилната си поддръжка на проекти с отворен код, което привлича много разработчици и фирми. Благодарение на силната си позиция в индустрията, GitHub често се възприема като стандартен избор за екипно сътрудничество и управление на проекти.



Фиг JJD. Всички репозитории в организацията на проекта.

За организацията на репозиториите е подходено по имплементация на микро сервиз. Тоест когато има нов микро сервиз или част от системата, се създава нова репозитория. Така кодът за една услуга остава изолиран в отделен проект, както е по модела на разпределените системи. Енкапсулирайки всеки сервиз, това позволява индивидуална настройка на репозиторията спрямо имплементацията. Така и за всяка репозитория се отделя и автоматизацията.

С този подход, всеки сервиз може да има своя специфична интеграция с инструменти за CI/CD, което позволява по-гъвкаво и независимо управление на процесите по разработка и доставка. Когато възникнат промени или подобрения в един микро сервиз, те не засягат директно останалите части на системата. Това улеснява мащабирането и поддръжката на системата, като всяка репозитория може да се оптимизира спрямо нуждите на съответния сервиз, без да се компрометира цялостното решение.

Когато има различни репозитории, те могат да се организират спрямо кодовите практики на съответния език, като се следват най-добрите стандарти за всяка технология. В зависимост от езика, разработчикът настройва репозиторията така, че да поддържа специфичната структура на проекта и инструментите, които са характерни за този език.

Например, за Java се използва структурата на директориите с разделение на папките src/main/java и src/test/java за основния код и тестовете. Интеграцията с инструменти като Maven или Gradle за управление на зависимости и изграждане на проекта е стандартна практика. Всяка Java репозитория може да има специфични правила за форматиране на кода с помощта на Checkstyle или PMD.

За JavaScript, репозиторията обикновено включва конфигурационни файлове като package.json за управление на зависимости чрез NPM или Yarn. Също така, ESLint се използва за поддържане на кодов стил и качество. Проектите могат да бъдат структурирани в зависимост от рамките, като например използване на директории src и dist при работа с React или Node.js.

При Rust, стандартната организация на репозиторията включва използването на Cargo.toml за управление на зависимостите и настройките на проекта. Структурата на папките включва директориите src за основния код и tests за тестовете. Кодовите практики се спазват с помощта на инструменти като rustfmt и clippy за формат и статичен анализ.

Всеки език предлага различни инструменти и подходи за автоматизация и контрол на качеството, които могат да се интегрират в съответните репозитории, за да отговарят на стандартите на екипа и проекта.

# Автоматизация

Автоматизацията представлява процесът на използване на технологии за извършване на задачи с минимална човешка намеса. В съвременния контекст, автоматизацията обикновено се отнася до софтуерни и хардуерни системи, които автоматизират рутинни и повторяеми дейности, подобрявайки ефективността, точността и скоростта на изпълнение.

### Основни цели на автоматизацията

**Повишаване на ефективността**: Автоматизацията елиминира нуждата от ръчно изпълнение на задачи, което значително намалява времето за изпълнение на процеси и увеличава производителността.

**Подобряване на качеството и точността**: Човешките грешки често водят до несъответствия и проблеми в качеството. Автоматизираните системи следват предварително дефинирани инструкции, което гарантира висока степен на точност и консистентност.

**Намаляване на разходите**: Чрез автоматизация се намалява необходимостта от ръчен труд и свързаните с него разходи. Това включва както преките разходи за заплати, така и непреките разходи, свързани с грешки и последващи корекции.

**Осигуряване на мащабируемост**: Автоматизираните системи могат лесно да се мащабират според нуждите на бизнеса, без да се изискват значителни допълнителни ресурси.

Автоматизацията играе ключова роля в съвременния свят, като трансформира начините, по които се извършват бизнес процеси и технологични операции. Тя осигурява значителни предимства по отношение на ефективност, качество, разходи и мащабируемост. Внедряването на автоматизирани системи продължава да бъде стратегически приоритет за много организации, които търсят начини да подобрят своята конкурентоспособност и да отговорят на нарастващите изисквания на пазара.

Pipeline в софтуерното инженерство представлява последователност от автоматизирани процеси, които се изпълняват с цел разработка, тестване и внедряване на софтуерни приложения. Тези процеси обикновено включват етапи като компилация, тестване, изграждане на артефакти и разгръщане на софтуера в производствена среда.



Фиг. Диаграма представяща цялостния работен поток за всеки Spring application в проекта.

#### Основни компоненти на Pipeline

**Източник на код (Source Code)**. Този етап включва изтегляне на последния изходен код от системата за контрол на версиите, като например GitHub, GitLab или Bitbucket.

**Компилация (Build).** На този етап изходният код се компилира, за да се провери дали няма грешки в кода и дали той може да се превърне в изпълним артефакт. Това включва и инсталирането на зависимостите, необходими за проекта.

**Тестване (Test)**. Тестовете се изпълняват автоматично, за да се гарантира, че новите промени не водят до регресия или грешки. Тестовете могат да включват unit тестове, интеграционни тестове, функционални тестове и др.

**Изграждане на артефакти (Artifact Creation)**. След успешната компилация и тестване, се създават артефакти, като например JAR файлове, Docker образи или други изпълними файлове, които ще бъдат разгръщани.

**Разгръщане (Deployment)**. Артефактите се разгръщат в различни среди, като тестови, staging или производствени среди. Този етап може да включва и изпълнението на миграции на бази данни, конфигурационни промени и др.

**Мониторинг и обратна връзка (Monitoring and Feedback)**. След разгръщането, системата се наблюдава за потенциални проблеми или аномалии. Обратната връзка от мониторинг системите може да бъде използвана за подобряване на следващите итерации на pipeline-а.

#### Приложение на Pipeline

Pipeline-ите се използват широко в практиката на DevOps и непрекъсната интеграция и разгръщане (CI/CD) за автоматизация на жизнения цикъл на софтуерната разработка. Те позволяват на екипите да постигнат бързо и надеждно разгръщане на нови версии на софтуера с минимална човешка намеса.

**Непрекъсната интеграция (CI)**. Pipeline-ите автоматизират интеграцията на кода от различни разработчици, като гарантират, че всяка нова промяна се проверява автоматично и не нарушава работата на съществуващия код.

**Непрекъснато разгръщане (CD)**. Pipeline-ите автоматизират разгръщането на нови версии на софтуера в различни среди. Това позволява бързо и често публикуване на нови функционалности и поправки на грешки.

**Подобряване на качеството на софтуера**. Чрез автоматизирани тестове и проверки, pipeline-ите гарантират високо качество на софтуера, като откриват проблеми рано в процеса на разработка.

Pipeline-ите са неотменима част от съвременната софтуерна разработка, която позволява на екипите да автоматизират и оптимизират процесите по изграждане, тестване и разгръщане на софтуера. Те осигуряват надеждност, ефективност и мащабируемост, като същевременно спомагат за намаляване на човешките грешки и подобряване на качеството на софтуерните продукти.

## GitHub Actions

Първата стъпка при автоматизацията започва от репозиторията съдържайки програминия код. Без код няма приложение. Доставчика на хостинг на репоситория избран е GitHub. GitHub е устойчив доставчик отличил се е на пазара от дълги години. GitHub Actions е платформа за автоматизация на работни потоци, интегрирана в GitHub. Тя позволява на разработчиците да автоматизират процесите по изграждане, тестване и разгръщане на техния код директно в хранилищата им. Това се постига чрез използване на "действия" (actions), които представляват индивидуални задачи, дефинирани в YAML файлове, наречени "работни потоци" (workflows).

### Основни характеристики на GitHub Actions

1. **Автоматизация на CI/CD**: GitHub Actions поддържа Continuous Integration (CI) и Continuous Deployment (CD), което позволява на разработчиците автоматично да тестват и разгръщат своя код при всяка промяна.
2. **Множество среди за изпълнение**: Платформата предоставя възможност за изпълнение на действия в различни среди, включително Linux, macOS и Windows.
3. **Гъвкавост и мащабируемост**: GitHub Actions поддържа паралелно изпълнение на задачи и позволява създаването на сложни работни потоци с условни логики, които могат да се мащабират в зависимост от нуждите на проекта.
4. **Интеграция с GitHub**: Платформата е дълбоко интегрирана с GitHub, което улеснява управлението на работните потоци директно от хранилището на проекта и използването на събития като комити, pull requests и издания като тригери за изпълнение на действия.

### Работен поток използван в проекта

Работният поток, дефиниран по-долу, автоматизира процеса на изграждане на проект с Gradle и качване на резултатните артефакти в хранилището. Този работен поток е настроен да се изпълнява при всеки push или pull request към основния клон на хранилището.

Дефиниция на работния поток



Фиг. IXI. Скриптов код използван за github action в репозитория volts-server

#### Обяснение на работния поток

**Име на работния поток**:

name: Gradle-build

Работният поток е именуван "Gradle-build", което указва, че основната му цел е изграждане на проект с Gradle.

**Тригери за изпълнение**:

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

Работният поток се изпълнява при всеки push или създаване на pull request към основния клон ("main") на хранилището.

**Дефиниране на работа**:

jobs:

build-without-cache:

runs-on: ubuntu-latest

Определя се една работа с име "build-without-cache", която ще се изпълнява на най-новата версия на Ubuntu.

**Стъпки на работа**: Работният поток съдържа няколко стъпки, които се изпълняват последователно:

**Изтегляне на кода**:

yaml

Copy code

- uses: actions/checkout@v2

Тази стъпка използва предварително дефинирано действие за изтегляне на кода от хранилището.

**Настройка на JDK 17**:

- name: Set up JDK 17

uses: actions/setup-java@v2

with:

java-version: 17

distribution: 'adopt'

Тази стъпка настройва JDK версия 17, използвайки AdoptOpenJDK дистрибуцията. Това е същото JDK което се използва в конфигурационния файл на главния проект. Спазва се еднаквост при всичко скриптове въпреки че са различни скриптове, които се сами по себе си се изпълняват на различни машини или виртуални машини.

**Даване на права за изпълнение на gradlew**:

- name: Grant execute permission for gradlew

working-directory: volts-server

run: chmod +x gradlew

Тази стъпка дава права за изпълнение на скрипта gradlew.

**Изграждане с Gradle**:

- name: Build with Gradle

working-directory: volts-server

run: ./gradlew build

Тази стъпка изпълнява командата за изграждане на проекта с Gradle.

**Изброяване на файловете в директорията за изграждане**:

- name: List build directory

working-directory: volts-server

run: ls -la build/libs

Тази стъпка изброява съдържанието на директорията build/libs, за да покаже изградените артефакти.

**Качване на JAR файла**:

- name: Upload JAR file

uses: actions/upload-artifact@v3

with:

working-directory: volts-server

name: my-app-build

path: build/libs/\*.jar

Тази стъпка качва изградените JAR файлове като артефакти на работния поток.

**Комитване на JAR файла в хранилището**:

- name: Commit JAR file to repository

working-directory: volts-server

run: |

git add build/libs/\*.jar

git commit -m "Add build artifacts"

git push origin main

env:

GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

Тази стъпка комитва изградените JAR файлове в хранилището, използвайки конфигурирано GitHub потребителско име и имейл. Тук се използва вътрешната променлива за пазене на таен текст secrets.GITHUB\_TOKEN. Проекта представлява public repository тоест всеки може да го погледанe, да го изтегли и да предложи промени по кода. Затова когато имаме секретни ключове за достъп до личните ресурси на проекта те трябва да се пазят в тайма. Затова платформата github предоставя възможността за скриване на ценна информация под формата на secret. В този случай е създадена тайна която пази ключ за достъп до ресурсите на проекта. Github само по себе си когато срещне подобна тайна при изпълнение ще бъде подменена с тайният текст.

Този работен поток илюстрира как GitHub Actions се използван за автоматизация на процесите по изграждане и публикуване на артефакти в софтуерен проект. Автоматизирането на тези процеси не само спестява време и усилия, но и гарантира последователност и надеждност в цикъла на разработка. В проекта е използван за всяка репозитория github actions поради лекотата от използването им и екосистемата на github.

### Работен поток за публикуване на npm пакет

Работният поток, дефиниран по-долу, автоматизира процеса на изграждане и публикуване на npm пакет при всяка промяна в основния клон (main) на хранилището. Това гарантира, че всяка нова версия на пакета се публикува автоматично в npm регистъра, без нужда от ръчно изпълнение на стъпките. Този работен поток е ключов за поддържане на актуални версии на софтуера инсталиран върху конекторите. Когато пакета е качен на облака може лесно потребителя да погледне активната версия и да я обнови. Тук инсталирането на нова версия зависи от потребителя тъй като веднъж инсталирайки и монтиране на устройството няма пряк достъп до него. Често един софтуер ако е отрит проблем той сам по себе си ще съобщи за грешка и нуждата от обновяване към нова версия. Когато е пакетиран кода и готов за инсталация в облака това се случва лесно. Затова ни е нужен този работен поток. За да може автоматично при излизане на нова версия да може да се пакетира и качи на облака. Така и улеснява създаването на нови дистанционни конектора.



Фиг. XIIX. Скрипт за работен потока пакетиращ и публикуващ към NPM облак

#### Обяснение на работния поток

**Име на работния поток**:

name: Publish npm package

Работният поток се нарича "Publish npm package", което указва, че основната му цел е публикуване на npm пакет.

**Тригери за изпълнение**:

on:

push:

branches:

- main

Работният поток се изпълнява при всяко push събитие към основния клон ("main") на хранилището.

**Дефиниране на работа**:

jobs:

build:

runs-on: ubuntu-latest

Определя се една работа с име "build", която се изпълнява на най-новата версия на Ubuntu.

**Стъпки на работа**: Работният поток съдържа няколко стъпки, които се изпълняват последователно:

* + **Изтегляне на кода**:

- name: Checkout code

uses: actions/checkout@v3

Тази стъпка използва предварително дефинирано действие за изтегляне на кода от хранилището.

* + **Настройка на Node.js**:

- name: Setup Node.js

uses: actions/setup-node@v3

with:

node-version: "16" # Specify the Node.js version

Тази стъпка настройва Node.js версия 16, което е необходимо за изпълнение на npm команди и скриптове.

* + **Инсталиране на зависимости**:

- name: Install dependencies

run: npm install

Тази стъпка инсталира всички зависимости, дефинирани във файла package.json на проекта.

* + **Изграждане на пакета**:

- name: Build the package

working-directory: volts-connector

run: npm run build # Make sure your build script is defined in package.json

Тази стъпка изпълнява скрипта за изграждане на проекта, дефиниран в package.json файла. Работната директория е зададена на "volts-connector".

* + **Публикуване в npm**:

- name: Publish to npm

working-directory: volts-connector

run: npm publish

env:

NODE\_AUTH\_TOKEN: ${{ secrets.NPM\_TOKEN }}

Тази стъпка публикува изградените артефакти в npm регистъра. За аутентикация се използва секретният токен (NODE\_AUTH\_TOKEN), съхраняван в GitHub Secrets.

Този примерен работен поток илюстрира как GitHub Actions се използва за автоматизация на процесите по изграждане и публикуване на npm пакети. Автоматизирането на тези процеси осигурява последователност и надеждност при разпространението на софтуерни компоненти, като същевременно спестява време и усилия за разработчиците. Внедряването на подобни автоматизирани системи е ключов аспект на съвременната софтуерна разработка и допринася за по-ефективно управление на версиите и разпространението на софтуерни продукти.

# Комуникационен портал на приложението

За да се достъпи приложението от различните интерфейси то трябва всяка заявка да премине през комуникативния портал. Тапи методология е избрана поради множествено положителни показатели. Тези положителни показатели включват: единен вход което води до единни проверки за коректност, превод от rest метода на комуникиране по-бързия разширен протокол за опашка за съобщения( Advanced Message Queuing Protocol- AMQP)

# Защитен сервиз

## Автентикация

Автентикирането на потребители се случва на ниво защитен сервиз и включва потвърждаване на потребителя чрез емайл/потребителски име и парола. След което се създава JWT( Json Web Token) токен. За да се случи това потвърждаване на потребителската самоличност първо трябва да се приемат неговите данни и това е възможно чрез следния бизнес обект:

public record UserLogIn(String credentials, String password) {}

Който съдържа и поле за идентификация и поле за парола. Което поле за идентификация представлява или емайл или потребителско име и поред вина на полета се приемат два различни начина на идентифициране. При приемате на данните те минават през верифициране на типа на данните, това се отнася за всяка заявка към сървъра. При

## Авторизация

Авторизацията в уеб приложения се използва за установяване на идентичността на потребителите и определянето на техните права и разрешения за достъп до определени ресурси или функционалности в приложението. Авторизацията осигурява защита на чувствителните данни и ресурси, като позволява само на упълномощени потребители да ги достъпват. Системата за авторизация определя кои потребители имат право да използват определени функционалности или да виждат определени данни в приложението. Авторизацията позволява на приложението да запази информацията за влизането на потребителя в рамките на сесията, което дава възможност за персонализирани функционалности и оптимизация на потребителския опит. Системата за авторизация поддържа аудитни логове, които записват дейността на потребителите в приложението, като това помага при проследяване на проблеми или съдейства при изследване на сигурностни нарушения. Авторизацията играе ключова роля в управлението на идентичността на потребителите, включително аутентикацията, управлението на пароли и обновяването на правата за достъп. Общо казано, авторизацията в уеб приложения е важен механизъм за сигурност и управление на достъпа, който осигурява защита на данните и контролира потребителския достъп до ресурсите на приложението.

## OAuth

OAuth 2.0 е индустриалният стандартен протокол за оторизация. OAuth 2.0 се фокусира върху опростеността на разработчиците на клиенти, като същевременно предоставя специфични потоци за оторизация за уеб приложения, настолни приложения, мобилни телефони и устройства за всекидневна. Тази спецификация и нейните разширения се разработват в рамките на IETF OAuth Working Group. Протоколът за уеб авторизация (OAuth) позволява на потребителя да предостави a достъпът на уеб сайт или приложение на трета страна до защитения потребител ресурси, без непременно да разкриват своите дългосрочни пълномощия, или дори самоличността им. Например сайт за споделяне на снимки, който поддържа OAuth, може да позволи на своите потребители да използват мрежа за печат на трета страна сайт за отпечатване на личните им снимки, без да позволява отпечатването сайт, за да получите пълен контрол върху акаунта на потребителя и без да имате потребител споделя дългосрочните идентификационни данни на своите сайтове за споделяне на снимки със сайтът за печат. Тоест употребата му за приложението ще е приложимо при вписване от трета страна като на пример Google акаунт или Microsoft акаунт. Това улеснява първоначално навлиза в приложението. Така клиента има едно по малко препятствие за регистриране и употребяване на приложението. В днешните дни където съществуват множествено приложения, които вършат множествено услуги, се конкурира за вниманието на клиента си. Повече внимание към едно приложение се транслира директно към повече приходи за това приложение. И когато има нисък праг за приемане на ново приложение то има по-голям шанс да останат и да употребяват приложението. Така се подхожда по-тактично към приветстването на нов потребител. Пакетът протоколи OAuth 2.0 вече включва

* процедура за позволяване на клиент да се регистрира с разрешение сървър,
* протокол за получаване на токени за оторизация от оторизация сървър със съгласието на собственика на ресурса и
* протоколи за представяне на тези токени за оторизация на protected ресурси за достъп до ресурс.

Използвайки един протокол предоставя множествено функционалности. Така се установява единен работен протокол за безопасност на данните. Когато има единен протокол за безопасност предоставя и че помежду екипи и програмисти има един стандарт, който трябва да следва, осланявайки работния процес. Протокола предоставя и създаването на множествено токени за персистиране на сесии с клиента. За приложението е употребен стандарта JWT(Json Web Token) . Извора за използване на JWT токени лежи в факта че е универсален и лесен начин за менажиране на сесии помежду сървъра и потребителския интерфейс ( Фиг. №: LXL). Както е разгледано в следващата под точка, се знае че съхраняването на тези токени е реализирано в база от данни тип библиотека, която сама по себе си е изключителна бърза да вземане и писане на данни. Това означава че менажирането на сесии е изключително бързо поради избраните технологии за съхраняване и автентикация и авторизация.



Фиг. №: LXL- Изображение на как е реализиран модела за създаване на потребителски токен.

Криптиране на пароли

## База от данни отговорна за потребителски сесии

За запазване сесиите на активните потребители в момента се използва база от данни тип библиотека( dictionary) или още позната като ключ със стойност( key value pair). И конкретно тази база от данни тип библиотека е пряко и единствено сервиза със сервиза отговорен за защита(ink-security). Този сервиз както е разгледано в предишните подточки е отговорен за защитата на потребителските данни и общата безопасност на системата от нападения. Това не значи че другите сервизи не спазват традиционните практики за защита от атаки. Напротив те трябва да следват многослойни процедури и техники за избягване на масивни или централизирани атаки, като например DDOS( Denial-of-service attack). Но самите сервизи не трябва да отговарят дали дадената потребителска сесия е активна, това е работа на сервиза за защита който автентификация и авторизация потребителя да има достъп до дадените сервизи. Затова е нужна база която да съхранява времената информация за потребителя, неговата авторизация и до колко е валидна сесията.

Относно базата използвана за реализация на каширане на сесиите на потребителите е използвана дистанционно разгъната библиотечна база от данни Redis. Трябва да се уточни че се използва библиотечната функция на Redis, тъй като Redis поддържа и множествени други специализирани режими на съхраняване на данни. Други функционалности на Redis включват и: база от данни подходяща за търсачи (Search data base), съхраняване в стандарт JSON, граф бази от данни, таблична база от данни, клъстър база от данни и много други които могат да се комбинират и споделят информация помежду си. Но за целта на реализация на каширане на потребителски сесии сме използвали библиотечните функции на платформата. Redis е лидер относно този вид съхраняване на данни и е използван от най-големите софтуерни фирми да забързат свалянето на често използвани данни от традиционни релационни бази от данни. Като пример е социалната мрежа Twitter които използват Redis за да забързан предаване на най-популярните публикации на платформата.



Фиг. XX: Графика показваща цялостна бързина спрямо релационни и не релационни бази от данни.

В java се поддържа свързаност с базата от данни чрез библиотеката наречена ‚ jedis‘ и е създадена от същите създатели на базата от данни. Задава се нова инфомация към базата като след като първо се реализира връзка към базата от данни чрез:

*JedisPool pool = new JedisPool("localhost", 6379);*

След усъществена връзка се преминава към вземане, добавяне, променяне или изтриване на хеш от базата. Пример за вписване на данни в базата:

*jedis.hset("user-session:123", hash);*

# Микро сервизи

## Сервиз за пренасочване на входни заявки ( Volts-gate)

## Сервиз за сигурност ( Ink-security)

## Сервиз за анализ ( Volts-analytics)

## Сервиз за препоръки(Volts-recomendations)

Сервиза за препоръки служи за анализиране, изчисляване и връщане на бизнес ориентирани препоръки на клиента. Целта на сервиза е да бъде полезен на крайния потребител, като му препоръчва добри практики и централизирани съвети относно неговата продукция и разходи.

### Технология

Технологията зад имплементацията

### Имплементация

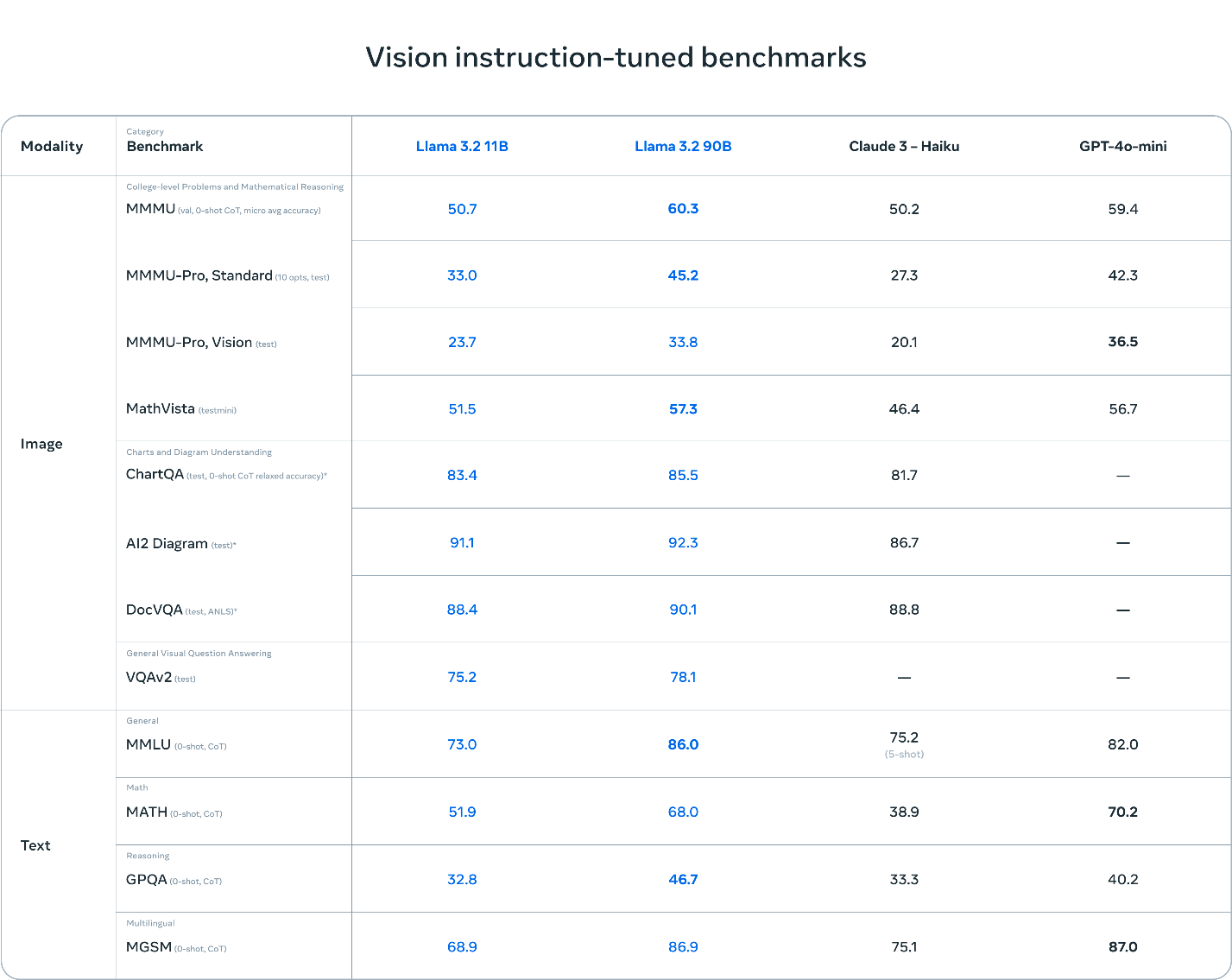
### Изкуствен интелект

Ollama е платформа с отворен код, създадена с цел да улесни използването на големи езикови модели (LLMs) на локални машини. Тя е разработена с акцент върху лесната инсталация и опростено управление на модели като Llama 2, Mistral и други. Ollama е насочена към разработчици, изследователи и ентусиасти, които искат локален контрол върху AI моделите си за задачи като генериране на текст, анализ на настроения или дори кодиране. Платформата е създадена с цел да бъде леснодостъпна и сигурна, като всички операции се изпълняват локално, без необходимост от облачни услуги, което предпазва потребителските данни и намалява разходите за използване на облачни ресурси.

Ollama се инсталира лесно на Linux сървъри чрез проста командна линия. За целите на поетка е стартирана на същата Linux машина както и всички контейнери. Самата инстанция на изкуствения интелект отнема 4 гигабайта от оперативната памет на машината, както и 90 гигабайта от пространство на харддиска.

Предимството на Ollama пред конкурентите като OpenAI е именно в това, че работи локално, което гарантира по-голяма сигурност на данните и елиминира зависимостта от външни сървъри. Докато OpenAI, с модели като GPT-4, е по-мощна платформа, тя изисква свързване към облак и често е по-скъпа, особено за по-малки компании. Ollama, от своя страна, предоставя по-достъпно решение, особено за потребители, които не разполагат с високопроизводителен хардуер или големи бюджети за облачни услуги.

Сред конкурентите на Ollama са и по-леки AI платформи като BERT и други модели с ниски хардуерни изисквания, но те често се нуждаят от повече ресурси и по-сложна настройка. Ollama предоставя по-лесна инсталация и използване, особено за по-неопитни потребители.



Фиг. НЙД. Оценка и сравнение на изкуствения интелект Llama по критерий свързани с текст и изображения, спрямо други модели изработени от Google и OpenAI.

 Фиг. НЙД. Оценка и сравнение на изкуствения интелект Llama по критерий свързани с конкретни задачи като общи знания, разсъждения, математика, използване на инструменти и многоезични умения., спрямо други модели изработени от Google.

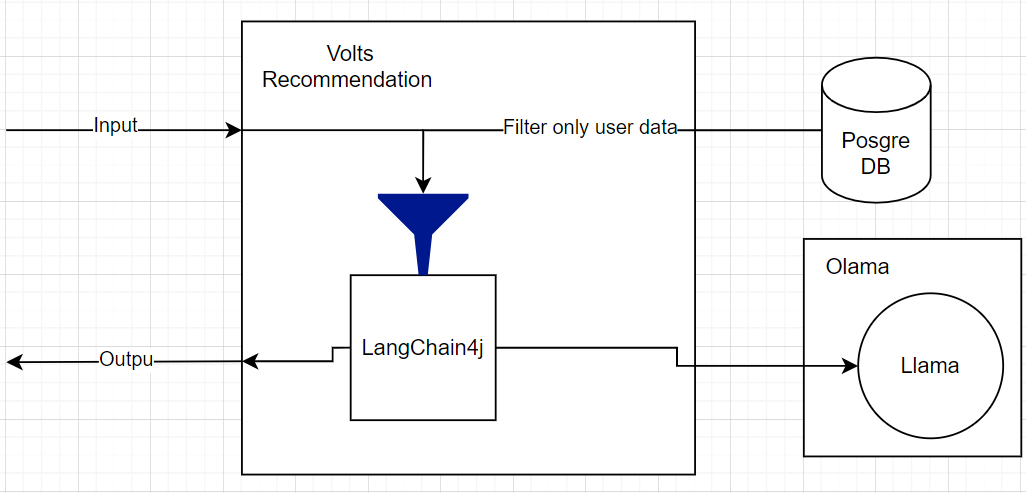
Имплементацията на сервиза за препоръки разчита главно на изкуствения интелект разгледан в предишния сервиз, както и на специализираната библиотека langChain4j. LangChain4J е библиотека за изкуствен интелект, създадена през 2023 година, като отговор на нуждата от подобна платформа за Java разработчици. Тя е вдъхновена от LangChain, популярна библиотека за Python, и има за цел да интегрира големи езикови модели (LLMs) и други AI технологии в Java приложения. Основната й роля е да свързва различни компоненти като LLM модели, текстови делители, изходни парсери и векторни хранилища, позволявайки създаването на персонализирани AI решения с минимален код.

LangChain4J предлага лесни за използване интерфейси и абстракции за работа с LLMs и поддържа интеграция с над 15 доставчика на AI модели, както и с много хранилища за векторно търсене и текстови модели. Разработчиците могат да създават приложения с поддръжка на обработка на естествен език и генеративен AI, както и да работят с мултимодални данни (текст и изображения).

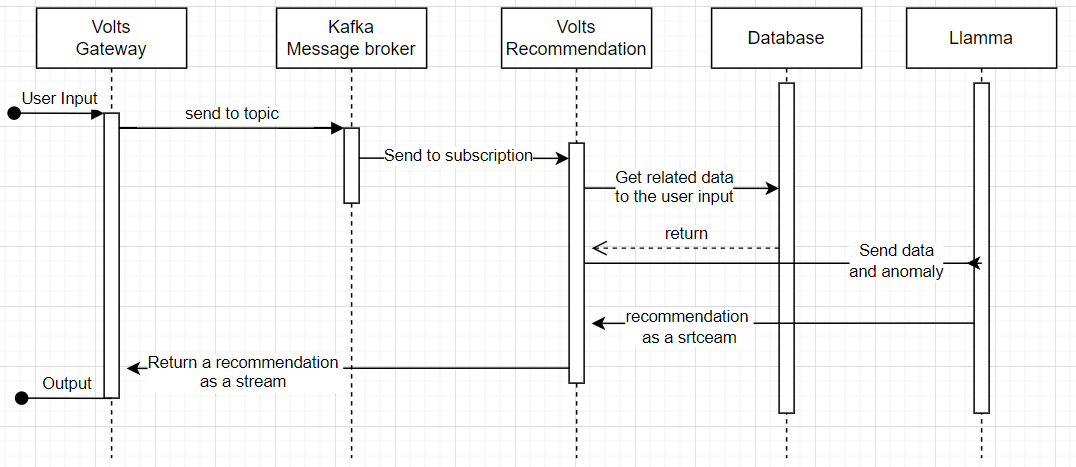
Сред предимствата на LangChain4J е фактът, че предоставя високо ниво на абстракция, което прави работата с LLM по-лесна за разработчиците, като скрива сложността на комуникацията с AI модели. Също така библиотеката предлага интеграция с Java фреймуърци като Spring Boot и Quarkus, което я прави подходяща за различни Java среди.

Конкурентни платформи като Haystack и LlamaIndex също предлагат подобни функционалности, но LangChain4J се откроява с по-добра съвместимост за Java и разширена поддръжка на различни модели и хранилища. Един недостатък, според някои разработчици, може да бъде, че библиотеката все още е в активна разработка и не всички функционалности са напълно завършени.

LangChain4J е идеална за разработчиците, които искат да създават AI приложения с Java без да навлизат в Python или други езици, и да използват AI услуги за специфични бизнес нужди. Точно затова е подбрана за използване в проекта. Библиотеката работи перфектно с Spring екосистемата която е избрана за проекта. Така лесно се имплементира абстрактен слой помежду клиента, неговите данни и изкуствения интелект. Нужна е този абстрактен слой, тъй като трябва безопасно да предадем само клиентските данни и нищо повече. Внимателно се постъпва когато всичките клиентски данни са в една база от данни и ги дели само един идентификационен номер. При липсата на подобен абстрактен слой, изкуствения интелект няма да открие разлика помежду данните на един и друг клиент. Затова употребата на междинен слой служи за ограничаване ресурсите на клиента.



Фиг. ККЛ. Структура на микро сервиза отговорен за препоръки. Възпроизведен като блок диаграма.



Фиг. ККЛ. Структура на микро сервиза отговорен за препоръки. Възпроизведен като диаграма на последователността.

Apache Log4j е популярна библиотека с отворен код, използвана основно в Java приложения за регистриране на системни поведения и събития. Тя е разработена през 1996 г. и широко се прилага както в малки, така и в големи проекти. Нейната цел е да улавя важни данни по време на изпълнение на приложенията за нуждите на дебъгинг, мониторинг и анализ. Тя използва йерархична структура на логерите, което позволява гъвкавост при регистриране на различни части от приложението. Log4j поддържа различни формати за изход и множество места за запис на логовете, включително файлове, бази данни и мрежови сокети.

Основните предимства на Log4j включват нейната ефективност, мащабируемост и гъвкавост, които позволяват на разработчиците бързо да диагностицират проблеми, по-лесно да поддържат приложенията и да оптимизират производителността. Въпреки това, както всяка логинг система, Log4j добавя известен товар на приложението, особено ако не е правилно конфигурирана. Лошо написан код за регистриране може да доведе до намаляване на производителността или ненужна сложност.

Разработчикът избира Log4j, защото тя поддържа различни изходни апендери, е нишково безопасна и ефективно управлява Java изключенията. Библиотеката също така е съвместима с други програмни езици като Python и C++ и се интегрира лесно с модерни инструменти за разработка като Maven и Gradle.

Въпреки своите предимства, Log4j има конкуренти в екосистемата на логинга, като SLF4J и java.util.logging, които могат да се предпочетат в определени ситуации поради различни набори от функции или по-проста конфигурация【34†source】【35†source】.

# Източници

Работа с база от данни Redis и java: <https://redis.io/docs/connect/clients/java/>

# Приложение

## Входни точки (End points)

### Сервиз за сигурност (Ink-security)

Вписване като потребител:

Post

/auth/..

Body:

{

‘username’:’user1’,

‘password’:’!pasword12345678’

}

Регистриране като потребител

Вземане на потребителски данни

Пропоръчване