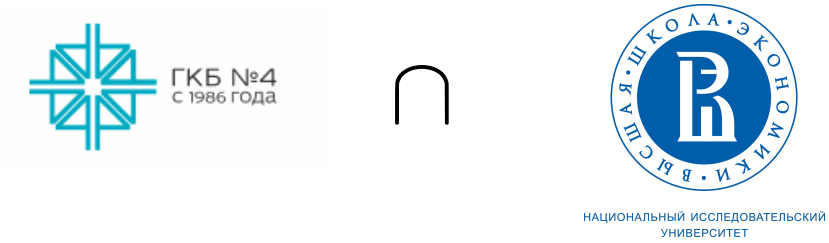
Общие сведения

Предпосылки

Наша научно учебная группа была сформирована на пересечении научных интересов нашей лаборатории и коллектива клиницистов-неврологов, которые занимаются исследованиями в области лечения острой фазы ишемического инсульта, реабилитации и профилактики вторичных инсультов.



Одна из ключевых целей - снизить долю смертности и инвалидизации за счет своевременного назначения эффективной профилактики вторичных инсультов. Актуальная доля криптогенных инсультов в практике наших коллег около 20%. При этом до половины из них возникают на фоне скрытой фибрилляции предсердий или нестенозирующего атеросклероза, т.о. создав систему умеющую различать хотя бы эти два подтипа инсульта мы в перспективе можем уменьшить долю криптогенных инсультов вдвое.

В основе идеи создания такой системы лежит предположение о различимости мрт-паттернов очага инсульта кардиоэмболического (на фоне фибриляций предсердий и схожих по патогенезу) и аэторотромботического подтипов. Предполагается, что можно обучить модель машинного обучения на имеющихся и ожидаемых данных, которая будет обладать ненулевым эффектом.

Бэклог продукта

Термин пришел из agile-практик, но для упрощения назовём **бэклогом** список желаемых возможностей и черт будущего продукта. Попробуем в этом стиле описать основные характеристики будущей системы.

1) **Доступность.**

Подразумевается что доступ к системе должен максимально простым, т.к. больший охват потенциально позволит получить больше данных для обучения и повысить качество модели. Т.е. чем меньше действий требуется со стороны пользователя и чем они проще тем лучше. Необходима доступность как с компьютера так и с мобильного телефона

2) **Низкий порог вхождения.**

Подразумеватеся что интерфейс должен быть лаконичным, а элементы управления должны быть максимально простыми и понятными. Нужно что бы охват на старте был как можно больше и данными могли без труда поделиться все пользователи которые ими обладают.

3) **Ассистирование врачу клиницисту.**

Система должна выдавать наиболее вероятный по её мнению тип инсульта, а так же указывать на рекомендуемые исследования.

4) **Изображения в качестве входных данных .**

В качестве входа система должна принимать изображения МРТ снимков (скриншоты или фотографии с экрана) с различимым очагом инсульта. Ответ должен выдаваться по одному или нескольким снимкам. В перспективе на вход модель должна иметь возможность принимать так же фотографии экг, кт снимки и другие медицинские показатели.

5) **Модульность.**

На ранних этапах развития системы, необходим только функционал отвечающий за сбор данных, при этом добавление остальных возможностей системе не должны приводить к существенным изменениям в других модулях или ломать логику их взаимодействия.

6) **Масштабируемость.**

На ранних этапах с небольшим количеством пользователей система может и не быть особо производительной, однако увеличение количества пользователей не должно становиться проблемой, т.е. замедление работы системы может отпугнуть пользователей.

7) **Гибкость и неприхотливость.**

Система должна быть легкобслуживаемой, легкоразворачиваемой и устойчивой. Должна быть возможность развернуть её на любом доступном сервере или облаке.

Технологии

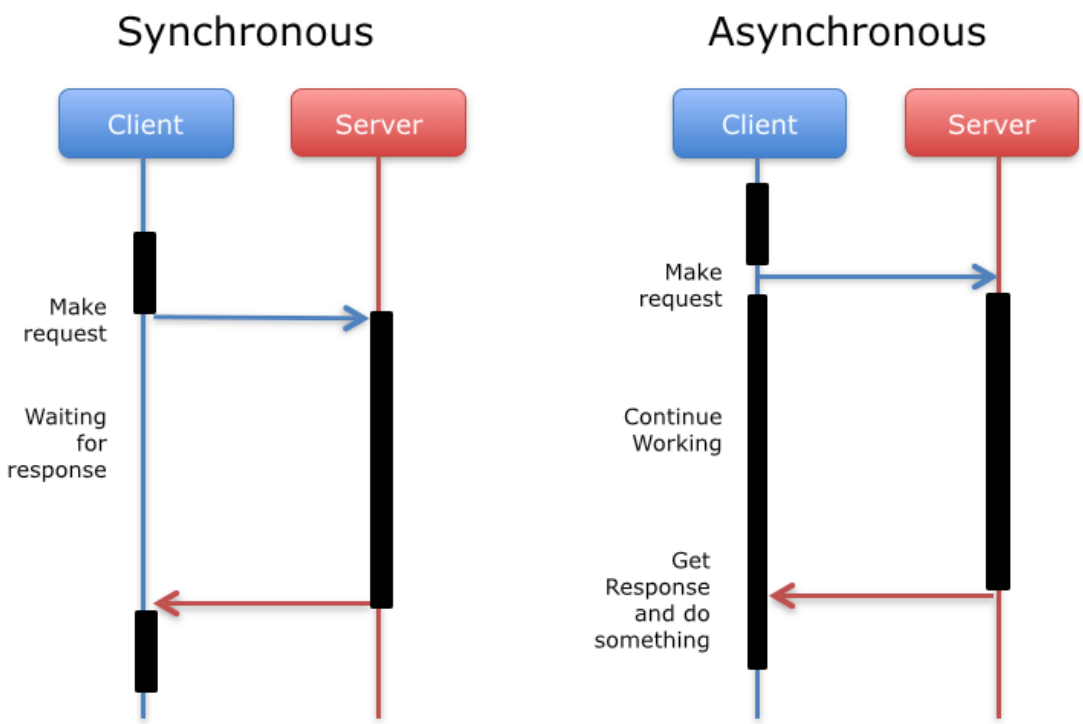
Опираясь на бэклог можно определиться с технологическим стеком. В качестве основного языка программирования выбран **Python** т.к. он наилучшим образом подходит для решения исследовательских задач в области постароения моделей глубокого обучения и более чем достаточен для решения всех остальных инфраструктурных задач (facebook, dropbox, instagram etc)

Основной фреймворк для построения моделей глубокого обучения - **PyTorch** библиотека разработанная facebook. Имеет удобный интерфейс и позволяет разработчику иметь максимальный контроль над моделью: задавать произвольную архитектуру, пользовательские слои, функции потерь. Совместно с **PyTorch Lightning** процесс обучения модели приобретает более стандартизированный характер (sklearn style) и упрощает использование вычислительных ресурсов (видеокарт, кластеров и т.д.).

```
import os
from torch import optim, nn, utils, Tensor
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
import lightning.pytorch as pl

# define any number of nn.Modules (or use your current ones)
encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))


# define the LightningModule
class LitAutoEncoder(pl.LightningModule):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        # it is independent of forward
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = nn.functional.mse_loss(x_hat, x)
        # Logging to TensorBoard (if installed) by default
        self.log("train_loss", loss)
        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=1e-3)
        return optimizer
```

```
# init the autoencoder
autoencoder = LitAutoEncoder(encoder, decoder)

# setup data
dataset = MNIST(os.getcwd(), download=True, transform=ToTensor())
train_loader = utils.data.DataLoader(dataset)

# train the model
trainer = pl.Trainer(limit_train_batches=100, max_epochs=1)
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

Для упрощения процесса доступа и взаимодействия с пользователем, вместо разработки собственного frontend интерфейса, было принято решение создать телеграм бота с использованим **telegram bot api**. В качестве фреймворка для разработки телеграм бота была выбрана асинхронная библиотека **Aiogram**. Асинхронность подразумевает конкурентное выполнение участков кода внутри программы, что позволяет повысить производительность в I/O-bound задачах. I/O-bound задачи - это задачи процесс исполнения которых упирается в ожидание ввода/вывода: ожидание подключения к серверу, к базе данных, ожидание ответа пользователя, ожидание ответа модели. Асинхронный код устроен таким образом что внутри программы есть главный цикл - eventloop внутри которого запускаются, приостанавливаются, продолжаются и завершаются корутины - функции которые имеют специальные команды отвечающие за передачу контроля исполнения наружу. Т.е. например такая функция может отправить запрос на подключение к базе данных и передать управление обратно в ивентлуп, и дожидаться двух событий: ответа от базы данных и возвращения управления внутрь функции, если оба эти события произошли исполнение функции продолжается дальше, если управление вернулось до ответа базы данных - корутина снова отдает управление главному циклу. Т.о. главный цикл поочереди дает возможность исполняться всем запущенным корутинам что позволяет добиться болеевысокой производительности чем в синхронном исполнении.

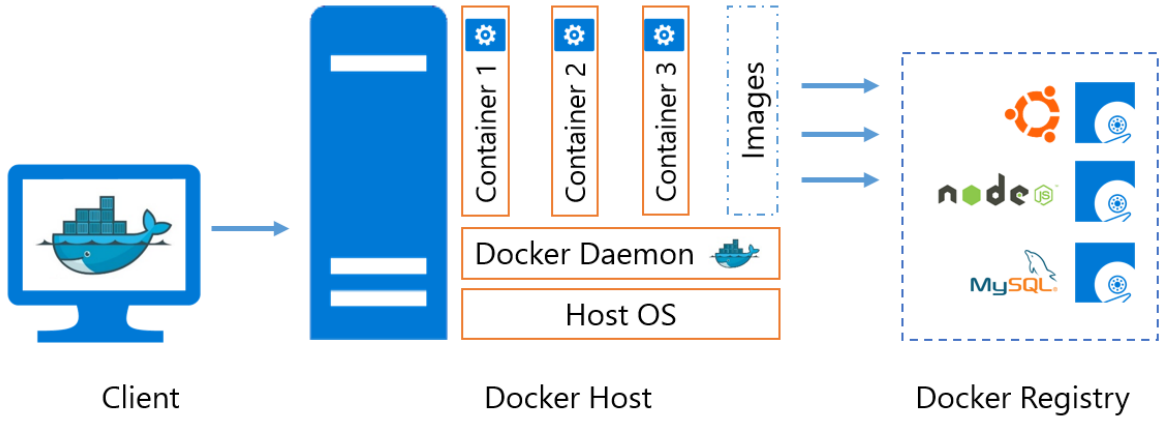


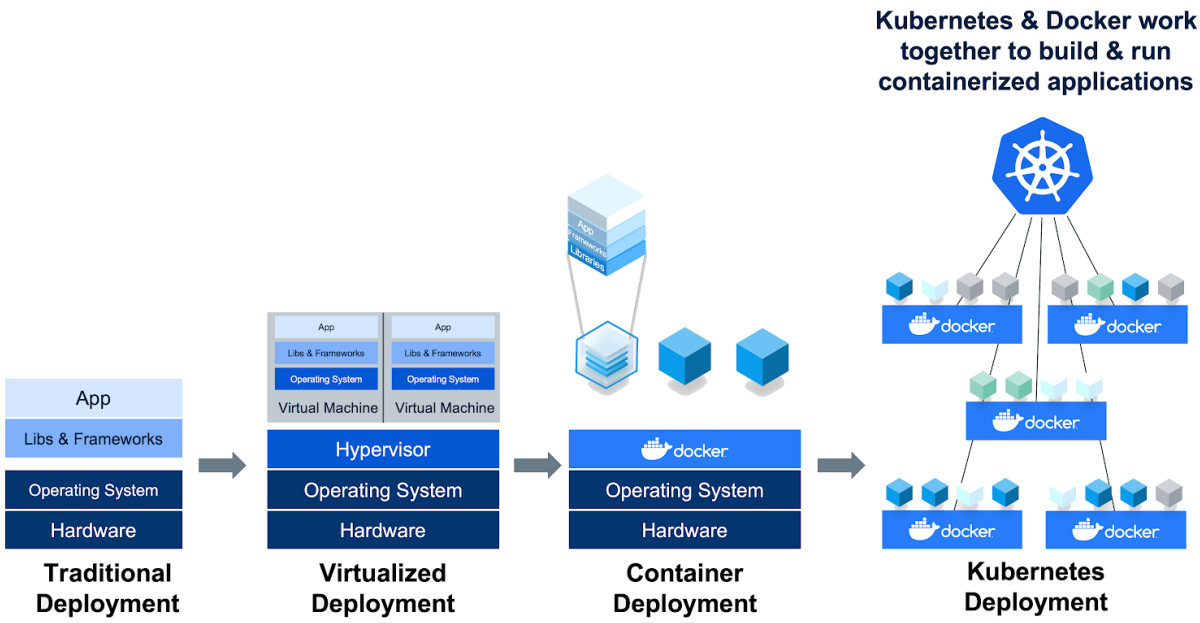
Логика использования асинхронной библиотеки для клиента, подразумевает что внутри клиента не должен использоваться синхронный cpu-boind код, т.к. он лишит нас всех преимущества. Cpu-bound задачи - задачи выполнение которых упирается в производительность вычислительных ресурсов, мощность процессора, видеокарты, кластера. Очевидно что расчеты внутри модели глубокого обучения - это cpu-bound задача, и следовательно её нужно вынести в отдельный модуль (сервис).

Вот здесь появлется разделение продукта на микросервисы. Микросервис - это самостоятельный квазипродукт, с собственной кодовой базой и стандартизированным интерфейсом взаимодействия выполняющий ограниченный набор действий. Микросервис - это абстракция упрощающая процесс разработки и сопровождения продукта. С точки зрения системного подхода, внутренее устройство микросервиса не имеет значения, важна лишь функция которую он выполняет. Микросервис может быть запущен на собственном устройстве и это не должно влиять на логику работы системы. Т.о. Микросервисная архитектура упрощает процесс развертывания особенно при использовании средств контейниризации.

Контейниризация - технология позволяющая абстрагироваться от зависимостей конкретного программного обеспечения. Идея контейниризации заключается в том что создается некоторая сущность - контейнер, которая

содержит в себе все необходимые компоненты для запуска установленного внутрь приложения. Это некое подобие самостоятельной “операционной системы” которое содержит собственную структуру каталогов, конфиги и зависимости. Однажды настроенный контейнер может быть запущен на любом совместимом устройстве. Конечно же лучший способ абстрагирования микросервисов друг от друга - это размещение каждого из них в своем контейнере. Многие стандартные микросервисы - такие как базы данных и брокеры сообщений поставляются от разработчиков в том числе ввиде контейнеров, которые нужно лишь сконфигурировать с помощью переменных окружения и можно использовать почти на любом сервере.



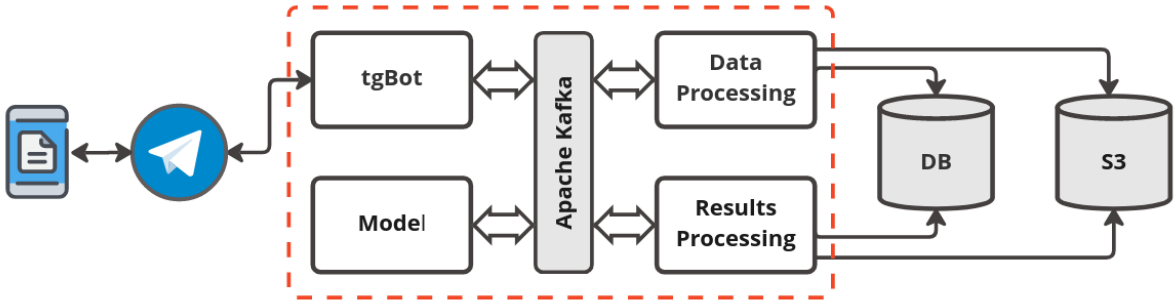


По скольку микросервисы - самостоятельные программы которые в общем случае могут быть запущены на разных устройствах, им нужно каким-то образом обмениваться данными. Для этих целей есть специальные другие микросервисы называемые **брокерами сообщений**. Вынося всю логику взаимодействия в отдельный микросервис мы упрощаем логику работы и уменьшаем объем необходимого кода каждого из микросервисов, потому что в нем остается достаточно описать логику взаимодействия с брокером сообщений. Брокеры сообщений в общем случае представляют из себя некоторую очередь или конвеер на который одни сервисы складывают сообщения, а другие читают. В нашей системе было принято решение использовать довольно развесистый брокер сообщений разработанный в linkedin - **Apache Kafka**. Кафка имеет возможность создавать топики - некоторые именованные области на которые одни сервисы складывают сообщения, а другие сервисы читают, при этом если топик читают несколько сервисов, то после прочтения группы сообщений, консьюмер - интерфейс кафки состороны сервиса получаеющего сообщения, делает коммит - помечает те сообщения которые он прочитал, т.о. кафка очень легко горизонтально масштабируется, добавив больше консьюмеров мы сможем параллельно обрабатывать группы сообщений из топика.

Для того что бы не грузить в через брокер сообщений сами изображения и ответы модели необходим еще один микросервис - объектное хранилище (такие используют для хранения больших объемов памяти например видео в youtube). В качестве S3 совместимого хранилища был выбран Minio.

Поскольку фотографи пользователь грузит не на прямую в бота а через сервер телеграм, то логику отвечающую за скачивание фотографий с серверов телеграм и их предварительную обработку разумнее вынести в отдельный микросервис, который так же будет работать в основном в синхронном режиме. Так же логику отвечающую за постобработку ответов модели и рассылку сообщений между пользователями логично вынести в отдельный микросервис.

Для хранения информации о пользователях и загруженных ими изображениях логично использовать реляционную базу данных - еще один микросервис. В качестве базы данных был вабран PostgreSQL благодаря большому сообществу и простоте использования.



tgBot

Интерфейсы:

- Сервер телеграм
- Брокер сообщений Кафка
- База данных Postgres

Data Processing

Интерфейсы:

- Сервер телеграм
- Брокер сообщений Кафка
- S3 хранилище

Model

Интерфейсы:

- Брокер сообщений Кафка
- S3 хранилище

Result Processing

Интерфейсы:

- Сервер телеграм
- Брокер сообщений Кафка
- S3 хранилище

- База данных Postgres

Kafka

Kafka Raft

Data Base

PostgreSQL

Simple Storage Service (S3)

Minio