# Как сделать быстрое и масштабируемое расчетное ядро на Python и не умереть рядом?

PiterPy 2025

# О себе

Иов Роман
Живу в г.Иркутск

Python-teamlead в Иркутской нефтяной компании последние 3 года, до этого – Сбер

Считаю, что автоматизация должна приносить деньги компании, а не наоборот

# План доклада

- Чуть-чуть вводной части, чем мы занимаемся
- Описание самой задачи
- Еще чуть профильной темы, чтобы понять контекст задачи
- С чего мы начинали
- Какие решения в итоге оказались приемлемыми
- Что в итоге получилось
- Куда двигаемся сейчас?

# Чем мы занимаемся?

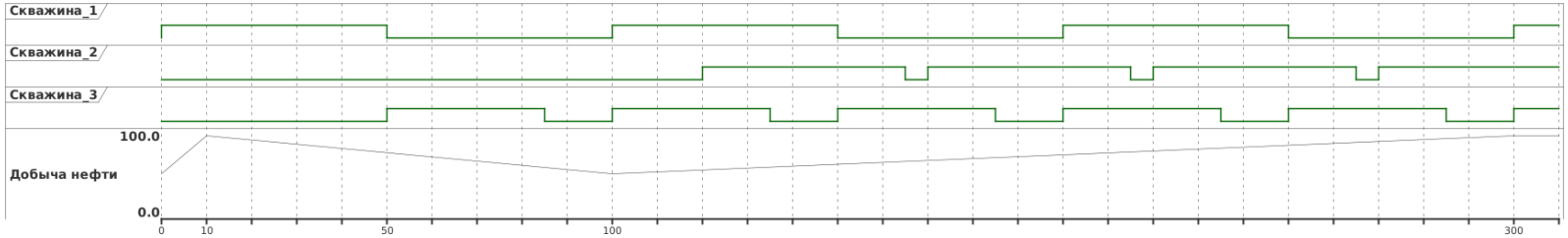
- Автоматизируем все, что связано с геологическим направлением в нашей компании
- Активно учим профильную тему (как оказалось без этого сложно)
- Нормализуем данные
- Строим новые процессы с использованием автоматизации
- Уничтожаем Excel

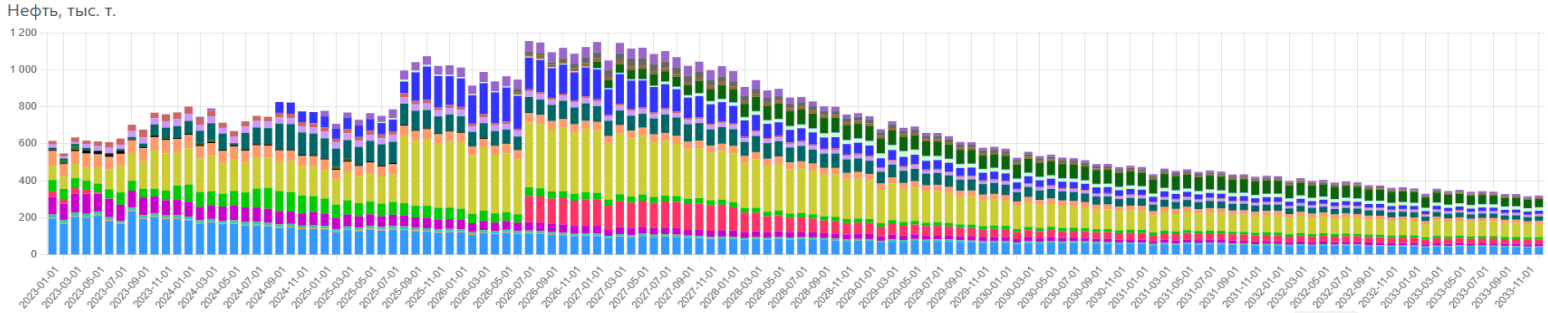
# Откуда берется нефть?

# Задача

Разработать информационную систему для моделирования прогноза показателей добычи нефти / воды / газа, с возможностью оперативного изменения прогноза (и миллион других требований...)

# Что это значит?



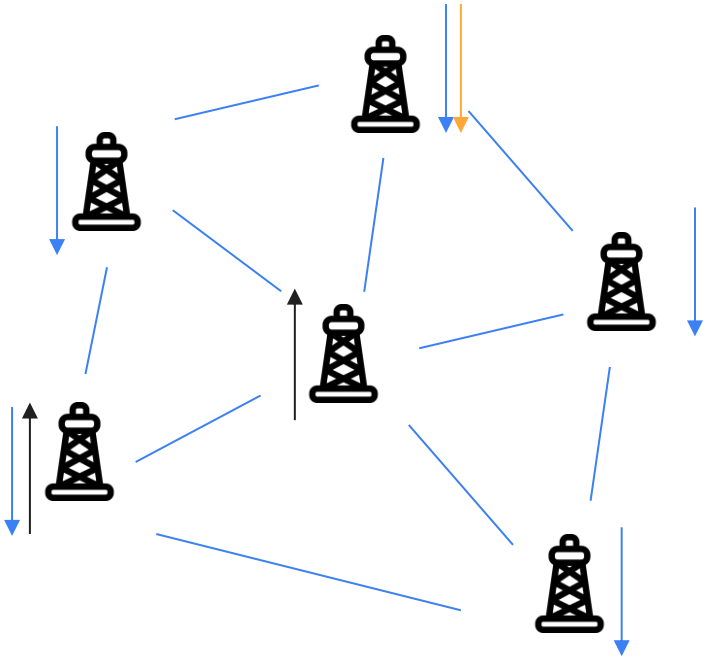
# Дополнительные требования

- Обеспечить оперативный пересчет – (5-10 секунд) – чтобы можно было менять прогнозы прямо на совещаниях
- Сделать набор базовых аналитик
- Обеспечить консистентность данных и снизить количество ошибок
- Считать прогноз по всей компании целиком – 2-3 тыс. скважин

# Почему задача сложная?

- Вся добыча предприятия состоит из суммы добычи по каждой скважины по отдельности в единицу времени (в нашем случае – ежемесячно)
- Период расчета – 20-50 лет.
- На скважинах проводят мероприятия как по повышению производительности, так и по понижению (и нужно еще считать эффект от каждого мероприятия)

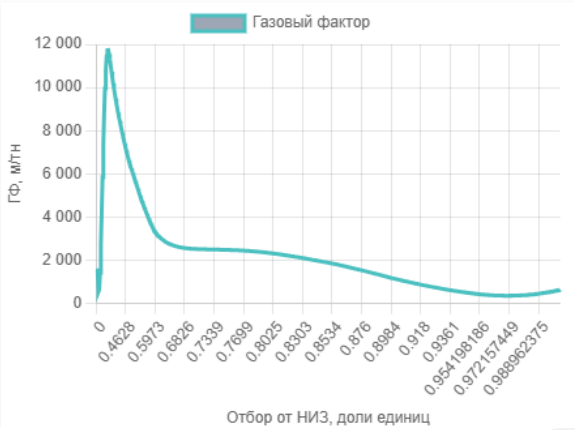
# Почему задача сложная v2?



- Скважины влияют друг на друга
- Скважины могут как добывать нефть, так и закачивать в обратную сторону воду и газ
- Могут остановиться, а могут запуститься
- Могут сначала добывать нефть, а потом закачивать воду

# Почему задача сложная v3?

- Поведение каждой скважины описывается набором геологических и физических характеристик, которые нужно интерполировать, нормировать и т.п.



# Выводы из предыдущих слайдов

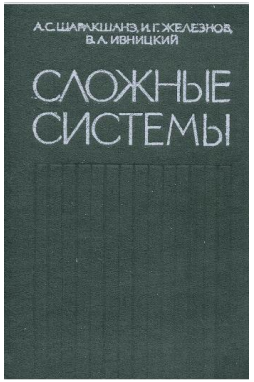
Коллега, который пытался сделать первый прототип, уволился через 3 месяца...

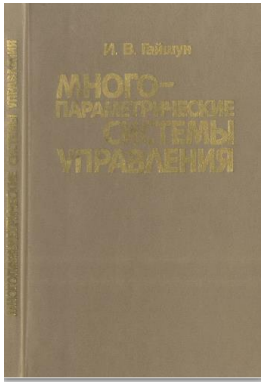
# Теперь серьезно

- Перед нами стоит сложная алгоритмическая задача
- Решение делаем на Python
- Без четкого планирования архитектуры мы утонем в легаси через пару месяцев
- Данные нужно сначала собрать, потом посчитать, а потом еще куда-то сохранить

# Что помогло в решении задачи?

- Как оказалось – читать книги – полезно!
- При решении помогли встать на путь истинный
- Гайшун И.В.  - Многопараметрические системы управления
- Ивницкий В.А., Железнов И.Г., А.С. Шаракшанэ – Сложные системы



# Входные данные

- Проведя оценку и посмотрев под разным углом, единственный способ представления – матричный (привет NumPy)
- На входе храним подготовленные SQL запросы к БД

```
@dataclass
class CalculationSendData:
    wells: callable = field(default_factory=lambda: WellsData.load_with_decline)
    gtm: callable = field(default_factory=lambda: GTMDeclinesData.load_all)
    gas_hat: callable = field(default_factory=lambda: GasHatData.load)
    # итоговые данные
    json_data: str = None
```

# Конфигурируем набор входных данных

- После получения параметров расчета – запускаем сбор данных из БД по необходимым доменным областям и собираем в json (чтобы передать в celery)

```python
def fill_json_data(self, calculation: Calculation):
        final_json_data = {}
        params_obj = CalculationFunctionParams(calculation)
        config_map = {
            "gtm": GtmConfig(calculation, params_obj),
        }

        for current_field in fields(self):
            config = config_map.get(current_field.name)
            if config and config.should_process():
                params = config.get_params()
                value = getattr(self, current_field.name)
                if value is not None:
                    final_json_data[current_field.name] = self.__get_data_from_db(value, calculation, params)
        setattr(self, "json_data", json.dumps(final_json_data, default=str))
```

# Формируем описание входных данных

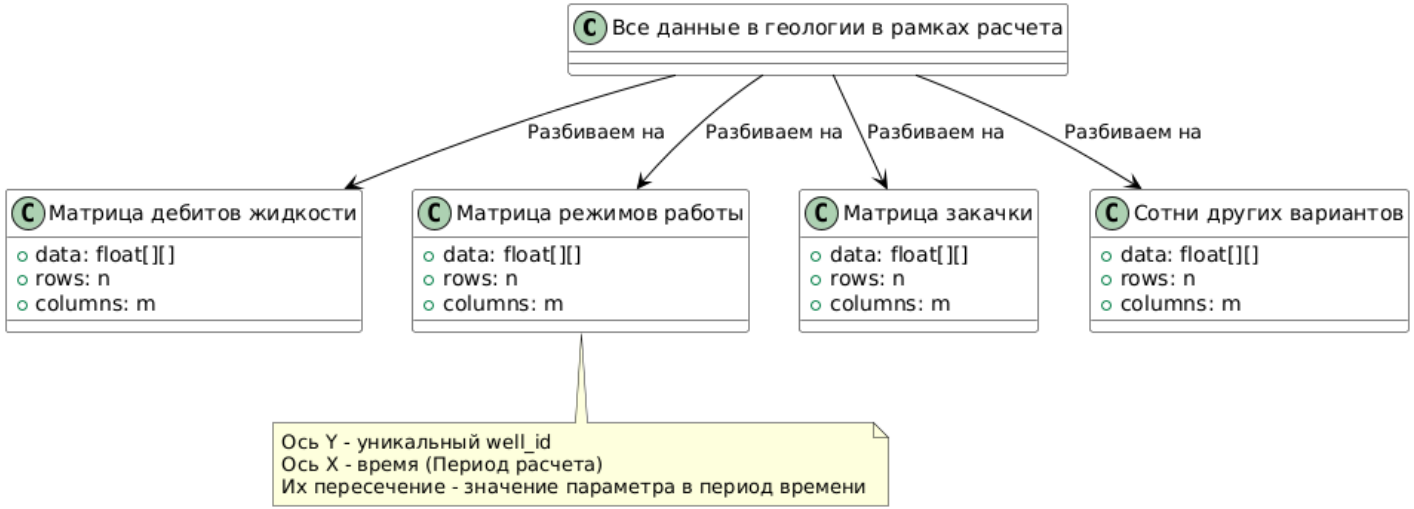
```python
@dataclass
class ColumnInfo:
    value: int
    type: Union[type[np.datetime64], type[np.int64], type[np.float64], type[np.object_], type[np.bool_]]


class AbstractColumnEnum(Enum):
    def __getattribute__(self, name: str) -> ColumnInfo:
        return super().__getattribute__(name)
```

```python
class WellsColumns(AbstractColumnEnum):
    WELL_ID = ColumnInfo(0, np.int64)
    NAME = ColumnInfo(1, np.object_)
    START_DATE = ColumnInfo(2, np.datetime64)
```

# Общая формула работы с данными

- Все данные в рамках расчета можно представить в виде 2-х мерных матриц (**у нас есть уникальный well_id**)
- Нужно привести все к виду, при котором мы не будем делать лишнего



# Этап 1. Инициализация

- Теперь нужно их распределить по слоям (классы, которые отвечают за специфичные области) и включить в единый config
- Слои делятся на базовые и опциональные.

```
# Создаем конфигурацию данных для расчета
(
    calculation_data_initalizer,
    optional_layers_initializer,
) = InitializerCreator.get_initializers(initial_data)
self.config.calculation_data = calculation_data_initalizer(**initial_data.__dict__)
# Инициализируем базовые расчетные слои
(
    self.config.base_layers,
    self.config.base_layers_data,
) = CalculatonsLayersInitializer.get(input_data=initial_data)
```

# Этап 1. Инициализация

- Как выглядит сборка инициализаторов классов?

```python
class OptionalLayersInitializer:
    OPTIONAL_INIT_LIST = {
        "injection": ClippedInjectionInitializer,  # учет только переводов в ППД
        "gtm": GTMLayerInitializer,
        "gtm_history": GTMHistoryLayerInitializer,
        "restriction": RestrictionLayerInitializer,
    }

    def __new__(cls, input_data: CalculationData, base_layers: CalculationLayers) -> OptionalCalculationLayers:
        optional_layer = OptionalCalculationLayers()

        # Инициализация слоев для исходных данных
        for key, initializer in cls.OPTIONAL_INIT_LIST.items():
            try:
                optional_layer.__setattr__(key, initializer(input_data, base_layers))
            except AttributeError:
                continue
```

# Этап 1. Инициализация

- Конкретная реализация слоя

```python
class GTMLayerInitializer:
    """
    Инициализатор слоя ГТМ
    """

    loader = GTMCalculatorLoader

    def __new__(cls, input_data: CalculationData, layers: CalculationLayers) -> GTMDataLayerBuilder:
        return GTMDataLayerBuilder(input_data.gtm)
```

```python
class GTMDataLayerBuilder:
    INIT_LIST: List[GTMParamCreator] = [
        GTMIdCountMatrix,
        GTMStartMatrix,
    ]
    def __init__(self, gtm_data: GTMProperty) -> None:
        self.gtm_wells_data: np.array = gtm_data.data
        self.columns_enum: GTMCalculatorAdapter = gtm_data.columns_list
        self.__validate()
```

# Как удобно срезать данные?

- Формируем свой набор преобразований для Enum, чтобы обеспечить последовательность индексов

```python
    @classmethod
    def transform(cls) -> Enum:
        dict_to_transform = {}
        for index, value in enumerate(cls):
            dict_to_transform[value.name] = index

        return Enum("Default", dict_to_transform)
```

```python
class WaterGasCalculatorAdapter(ExtendedEnum):
    WELL_ID = DeclinedWellDataColumnsList.WELL_ID
    WELL_OIL_DEBIT = DeclinedWellDataColumnsList.WELL_OIL_DEBIT
    WELL_OIL_CUMULATIVE_PRODUCTION = DeclinedWellDataColumnsList.WELL_OIL_CUMULATIVE_PRODUCTION
    WELL_GAS_CUMULATIVE_PRODUCTION = DeclinedWellDataColumnsList.WELL_GAS_CUMULATIVE_PRODUCTION
    WELL_WATER_CUMULATIVE_PRODUCTION = DeclinedWellDataColumnsList.WELL_WATER_CUMULATIVE_PRODUCTION
```

# Вопрос, зачем?

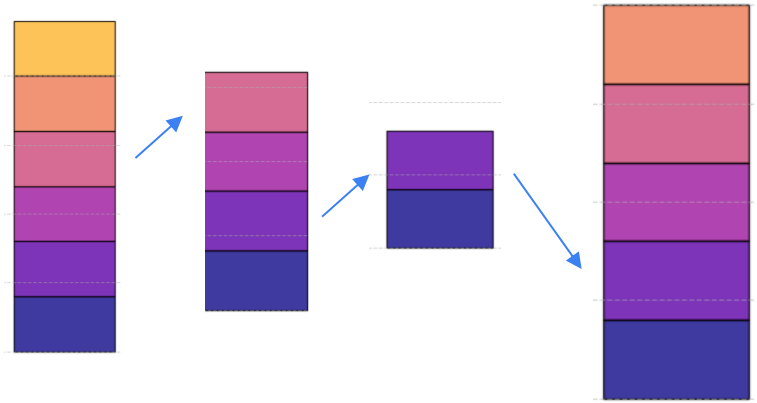
$$\begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & 30 \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & .. & a_{1n} \\ a_{21} & a_{22} & .. & a_{2n} \\ .. & .. & .. & .. \\ a_{m1} & a_{m2} & .. & a_{mn} \end{pmatrix} \end{array} \longrightarrow \begin{array}{c} \begin{array}{cc} 1 & 30 \end{array} \\ \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ .. & .. \\ a_{m1} & a_{m2} \end{pmatrix} \end{array} \longrightarrow \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ .. & .. \\ a_{m1} & a_{m2} \end{pmatrix} \end{array}$$

# Проблема размерностей

- Все было бы идеально, но…
- В зависимости от предметной области – матрицы данных в слоях будут разных размерностей

# Связи между слоями

- Добавляем следующую фичу из реляционных бд – связанные индексы
- Каждый слой хранит в себе ссылки на другие слои (двухмерную матрицу индексов)
- Да, мы тратим время на сопоставление (в нашем случае мы везде используем **well_id** – он всегда уникален)

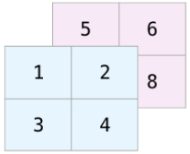


```
liquid_correction_coeff_transformed
= liquid_correction_coeff[config.optional_layers_data.gtm.id_matrix]
# Умножаем на полученный коэффициент
return calculated_liquid_debit * liquid_correction_coeff_transformed
```

# А где хранить каждый результат?

- Так как в некоторых случаях мы должны брать результат из предыдущих этапов – сохранение через двумерную матрицу
- Сохранение всегда через append в конец
- Нужно гарантировать размерность результатов (это делается через входные слои)

```
np.array([ [[1,2],[3,4]],
           [[5,6],[7,8]] ])
```




```python
class DefaultOutputLayer:
    def set_value(self, matrix: np.array) -> None:
        self.matrix = np.resize(matrix, (1, matrix.size))

    def add_horizontal_value(self, matrix_to_add: np.array) -> None:
        self.matrix = np.hstack([self.matrix, matrix_to_add])

    def add_value(self, matrix_to_add: np.array) -> None:
        self.matrix = np.vstack([self.matrix, matrix_to_add])

    def get_value(self) -> np.array:
        return self.matrix

    def set_matrix(self, matrix: np.array) -> None:
        self.matrix = matrix

    def set_last_column(self, values: np.array) -> None:
        self.matrix[-1] = values
```

# Этап 2. Подготовка расчетных функций

- Формируем набор расчетных модулей (опять же в зависимости от конфигурации расчета)

```python
class InitializableComputationModule(ABC):
    def calc(
        self,
        config: Config,
        output_layer: CalculationResult,
        index: int,
        month: datetime,
    ):
        raise NotImplementedError("Не реализован класс")

    def save(self, parameter: CalculationResultElement, data: np.ndarray) -> None:
        try:
            parameter.data.add_value(data)
        except AttributeError:
            parameter.data.set_value(data)
```

# Этап 2. Подготовка расчетных функций

- Расчетная функция – элемент single responsibility
- Выполняет только одну задачу, легко тестируется, работает только с конфигом и выходным слоем

```python
class GasFactorComputation(ComputationModule):
    @staticmethod
    def calc(
        config: Config,
        output_layer: CalculationResult,
        index: int,
        month: datetime
    ):
        factor = Gas.count_gas_factor_as_matrix(
            output_layer.passing_gas_production.data.get_value()[index],
            output_layer.month_oil_production.data.get_value()[index],
        )

        GasFactorComputation.save(output_layer.gas_factor, factor)
```

$$GOR = \frac{V_g}{V_o}$$

где: $GOR$ -- газовый фактор, $\frac{м^3}{м^3}$

$V_g$ -- объем газа, $м^3$

$V_o$ -- объем нефти, $м^3$

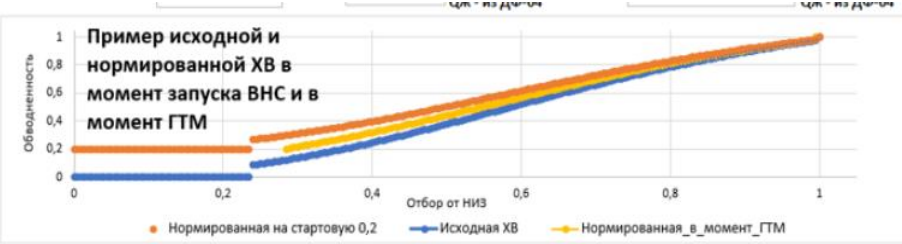
# Этап 3. Запуск расчета

- Итеративно и последовательно запускаем каждую расчетную функцию
- Параметрами является период расчета – сколько месяцев, столько и будет итераций

```python
def calculate(self) -> CalculationResult:
    self.__init_output_layer()
    # Формируем расчетный набор классов для проведения расчета
    computation_classes = ComputationPlanBuilder.build_configuration(self.config)
    for index, month in enumerate(self.config.base_layers.date_layer.period_row):
        # Класс, который контролирует обводненность и переводы
        self.config.base_layers.wells_switcher.watch(self.output_layer, self.config, index)
        for computation_class in computation_classes:
            computation_class.calc(self.config, self.output_layer, index, month)
    return self.output_layer
```

# Работа с линейными функциями

- А здесь проблема – простые математические функции работают быстро, но…
- SciPy – точно, но медленно
- NumPy – быстрее, но много кода и есть проблемы с **точностью**



# Работа с линейными функциями

- Грузим данные в json формате и связываем их со скважинами

```python
@dataclass
class DisplaceCharData:
    correlation_matrix: InterpolationCorrelationMatrix = None
    interpolate_char_list: List[List] = None


class DisplaceCharLayer(ScenaryConditionCharLayer):
    def _load_char(
        self,
        index,
        data: np.array,
        columns_list: DisplaceCharacteristicAdapter,
    ):
        char_json = json.loads(data[:, columns_list.DISPLACE_CHAR.value][index])
        if not char_json:
            raise ValueError
        return char_json
```

# Работа с линейными функциями

```python
class NumpyInterpolation:
    @staticmethod
    def interpolate(
        values_to_interpolate: np.array,
        correlation_object: InterpolationCorrelationMatrix,
        chars_list: list,
        min_value: float = None,
        max_value: float = None,
    ) -> np.array:
        values_to_interpolate = np.around(values_to_interpolate.astype(float), decimals=7)
        interoplated_list = []
        for indexes, char_indexes in zip(correlation_object.indexes_list,
correlation_object.char_index):
            interpolation_result = np.interp(
                values_to_interpolate[indexes],
                chars_list[char_indexes]["x"],
                chars_list[char_indexes]["y"],
                left=min_value,
                right=max_value,
            )
            interoplated_list.extend(interpolation_result)

        return np.array(interoplated_list, dtype=float)
```

# Этап 4. Завершение и сохранение

- Дошли до последней итерации – завершаем расчет

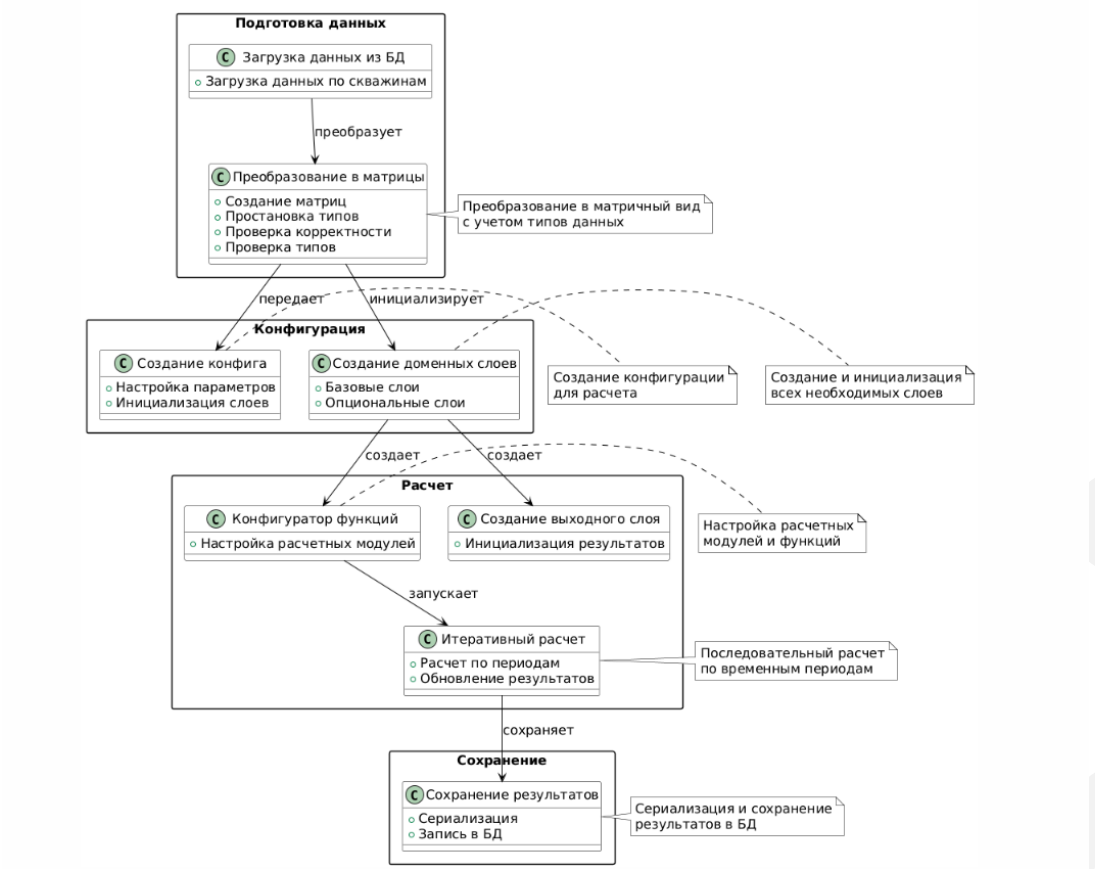
```
result = calculator.calculate()
data = pickle.dumps(result)
with transaction.atomic():
    try:
        CalculationResult.objects.filter(calculation_id=pk).delete()
    except CalculationResult.DoesNotExist:
        pass
    CalculationResult(calculation_id=pk, data=data).save()
```

# Почему сохраняем так?

- Заказчик желает быстро оценить результат расчета и быстро пойти пересчитать что-нибудь
- После того, как результат устраивает – запускается отдельный процесс с сохранением в реляционную бд (через timescale parallel copy)

| | id [PK] bigint | calculation_id bigint | data_size text | to te |
|---|---|---|---|---|
| 1 | 6349 | 1156 | 22 MB | 1 |
| 2 | 883 | 232 | 13 MB | 1 |
| 3 | 983 | 238 | 13 MB | 1 |
| 4 | 997 | 240 | 13 MB | 1 |
| 5 | 5917 | 1075 | 12 MB | 1 |
| 6 | 996 | 234 | 12 MB | 1 |
| 7 | 998 | 241 | 12 MB | 1 |
| 8 | 921 | 229 | 11 MB | 1 |
| 9 | 6002 | 1080 | 11 MB | 1 |
| 10 | 6260 | 1130 | 11 MB | 1 |

# Конечная схема работы ядра



# Что имеем сейчас?

- Ежедневно считаются различные варианты бизнес-плана
- Количество активных пользователей системы – 250 человек
- Время одного расчета на 2000 скважин – 5-7 секунд
- **Добавление нового функционала занимает минимальное количество ресурсов**
- В связке с Django наблюдаются проблемы с производительностью (вполне логично)

# Вывод

- Даже такие задачи можно решать в условиях ограниченных ресурсов
- Глаза боятся, руки делают
- Здесь должна быть мотивационная картинка

# Спасибо за внимание

Секция вопросов от слушателей

Если есть вопросы **-> @mickeyeasy**