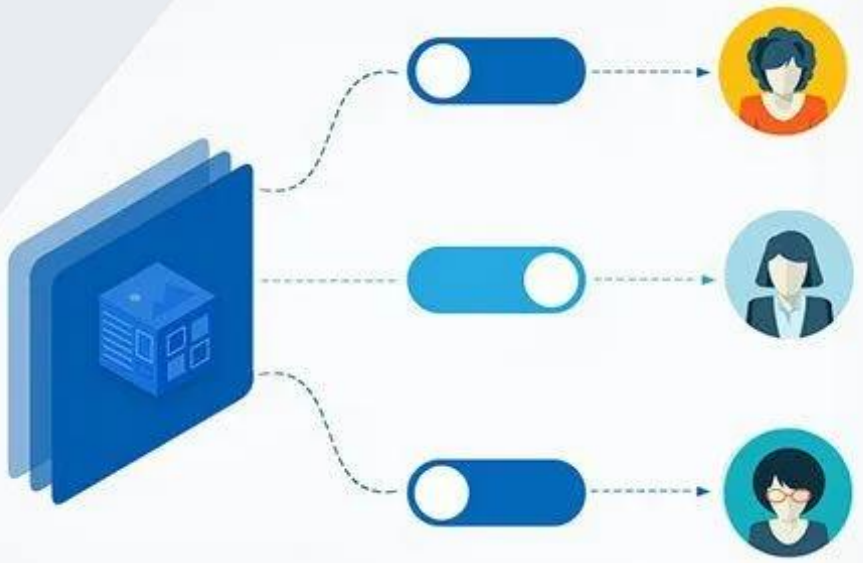
RUDN university

СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ ПАРАМЕТРЫ

КОНФИГУРАЦИЯ, ОКРУЖЕНИЕ, FEATURE TOGGLE



Config Management

КОД И КОНФИГУРАЦИЯ

Конфигурация — это внешний по отношению к коду слой, который меняет поведение системы без изменения алгоритмов.

Код задает множество возможных траекторий исполнения, конфигурация выбирает из них конкретную траекторию и подставляет значения параметров.

КОНФИГУРАЦИОННЫЕ ПАРАМЕТРЫ И ИХ РОЛЬ

Пример. Параметры окружения

```
16  OPENROUTER_API = "https://openrouter.ai/api/v1/chat/completions"
17  OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")
```

Код не меняется при смене ключа или кастомного прокси-URL

Конфигурация включает параметры доступа к внешним системам, количественные лимиты, настройки диагностики и флаги включения функций.

Эти параметры не добавляют новый код, а управляют режимами работы уже реализованных механизмов.

КОНСТАНТЫ В КОДЕ

Пример 1. Плохо

```
1   TIMEOUT_S = 5  # где-то в середине файла
2   r = requests.post(URL, json=payload, timeout=TIMEOUT_S)
```

Если таймаут нужно поднять до 10 секунд в проде, придется менять код и выкатывать новую версию

Пример 2. Очень плохо

```
1   r = requests.post("https://openrouter.ai/api/v1/chat/completions",
2                     json=payload, timeout=5)
```

URL и таймаут зашиты прямо в вызов. Для другой среды (другой API или прокси) придется редактировать исходник и надеяться, что везде заменили

Жестко зашитые константы в коде делают любое изменение параметров изменением исходников и релизом.

Такой подход быстро перестает быть управляемым и практически исключает корректную работу с разными окружениями.

КОНФИГУРАЦИОННЫЕ ФАЙЛЫ

Пример 1. YAML-конфиг

```yaml
openrouter:
  url: "https://openrouter.ai/api/v1/chat/completions"
  model_default: "tngtech/deepseek-r1t2-chimera:free"

limits:
  max_prompt_chars: 600
  max_response_tokens: 400
```

Код обращается к config["limits"]["max_prompt_chars"], не знает, откуда это взялось.

Пример 2. Разные файлы для разных окружений

config.dev.yaml

тестовые модели, LOG_LEVEL: DEBUG

config.prod.yaml

боевые модели, LOG_LEVEL: INFO, выше лимиты или ниже — по ситуации.

Конфигурационные файлы отделяют значения параметров от логики и позволяют менять настройки без редактирования кода.

Они становятся явной границей между алгоритмами и настройками, но требуют дисциплины в структуре, валидации и обновлении.

ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Пример. Настройка через окружение

```
environment:
  - OPENROUTER_API_KEY=${OPENROUTER_API_KEY}
  - LOG_LEVEL=INFO
```

Переменные окружения позволяют одной и той же сборке приложения запускаться с разными параметрами в разных средах.

Они особенно подходят для секретов и базовых параметров, но из-за плоской структуры обычно дополняются другими источниками конфигурации.

ЦЕНТРАЛИЗОВАННЫЕ КОНФИГУРАЦИОННЫЕ СЕРВИСЫ

Пример. Сервис конфигурации

Условный `config-service`

- по запросу `GET /config/bot-service?env=prod` возвращает JSON с параметрами
- наш бот при старте делает этот запрос и инициализирует настройки из ответа

Централизованный конфиг-сервис делает конфигурацию отдельной подсистемой с единым источником истины и аудитом изменений.

Он упрощает управление множеством компонентов, но усложняет архитектуру и сам становится критической зависимостью.

ОКРУЖЕНИЯ КАК КОНФИГУРАЦИОННЫЕ ПРОФИЛИ

Пример. Один код – несколько профилей

```
1    ENV = os.getenv("APP_ENV", "dev")
```

Конфиги:

`config.dev.yaml`, `config.prod.yaml`.

Файл выбирается по ENV, логика бота одна и та же.

DEV и **PROD** следует рассматривать как разные конфигурационные профили одной кодовой базы.

Отличия между окружениями должны задаваться конфигурацией, а не разными ветками исходного кода.

ОТЛИЧИЯ КОНФИГУРАЦИЙ **DEV, PROD**

Пример. Разные URL, ключи

DEV:

```
OPENROUTER_API_URL=https://openrouter.ai/api/v1/chat/completions
OPENROUTER_API_KEY=sk-test-...
ENABLE_EXPERIMENTAL_COMMANDS=true
```

PROD:

```
OPENROUTER_API_URL=https://openrouter.ai/api/v1/chat/completions
OPENROUTER_API_KEY=sk-prod-...
ENABLE_EXPERIMENTAL_COMMANDS=false
```

Окружения отличаются адресами и типами ресурсов, уровнем логирования, лимитами и включенностью экспериментальных функций.

Осмысленная система профилей конфигурации делает поведение окружений предсказуемым и воспроизводимым.

СТАТИЧЕСКИЕ НАСТРОЙКИ И ИХ СВОЙСТВА

Пример 1. Строка подключения к БД

```
DB_URL=sqlite:///bot.db
```

При изменении на `postgres://...` нужно перезапустить приложение, потому что пул соединений создается при старте

Пример 2. Включение/выключение механизма кэширования

```
ENABLE_REDIS_CACHE=true
REDIS_URL=redis://redis:6379/0
```

Если отключить кэш в проде, придется перезапустить сервис, чтобы не оставлять процессы, которые думают, что Redis есть

Статические настройки используются при инициализации и влияют на структуру внешних и внутренних связей системы.

Их изменение требует перезапуска и рассматривается как управляемая инфраструктурная операция.

ДИНАМИЧЕСКИЕ НАСТРОЙКИ

Пример. Динамические настройки

В таблице настроек:

```
setting_key      | setting_value
-----------------+--------------
max_prompt_chars | 600
```

Администратор меняет значение в таблице с **600** на **1000** — бот не перезапускается, все следующие запросы выполняются используя новое значение.

Старые запросы уже обработаны, ничего пересоздавать не нужно.

Динамические настройки могут меняться без рестарта и учитываются на уровне отдельных запросов или задач.

Параметр считается динамическим, если не приводит к неочевидному поведению и не требует пересборки приложения.

ПРОЕКТИРОВАНИЕ ДИНАМИЧЕСКИХ НАСТРОЕК

Пример 1. Чтение из переменных окружения

```python
def get_max_prompt_chars():
    return int(os.getenv("MAX_PROMPT_CHARS", 600))
```

Каждый вызов читает актуальное значение из окружения. Администратор меняет переменную — эффект мгновенный (но дорого, если много вызовов).

Пример 2. Периодическое чтение из БД

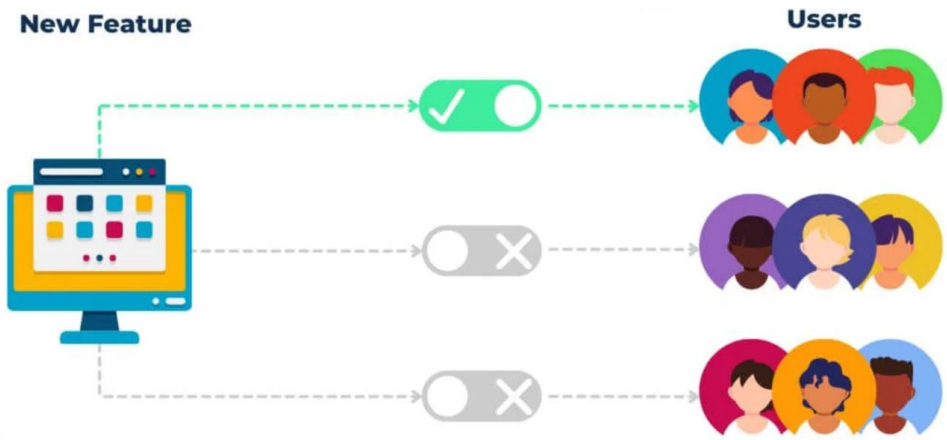
```python
_last_reload = 0
_max_prompt_chars = 600

def get_max_prompt_chars():
    global _last_reload, _max_prompt_chars
    if time.time() - _last_reload > 60:
        _max_prompt_chars = load_from_db("max_prompt_chars")
        _last_reload = time.time()
    return _max_prompt_chars
```

Настройка обновляется раз в минуту, нагрузка на БД ограничена.

Динамические настройки могут меняться без рестарта и учитываются на уровне отдельных запросов или задач.

Параметр считается динамическим, если не приводит к неочевидному поведению и не требует пересборки приложения.

FEATURE TOGGLES



Пример. Новая фича

```
1   if settings.enable_summary:
2       # код
```

Включаем фичу переключателем enable_summary в конфиге.
Если что-то пошло не так, отключаем обратно, не трогая код.

Feature Toggles управляют включенностью ветвей поведения и позволяют разнести по времени поставку кода и включение функции.

Они дают возможность быстро включать и отключать функциональность без релиза и упрощают поэтапный вывод новой функциональности.

FEATURE TOGGLES: ТИПЫ

Пример 1. Глобальный флаг

```
feature_flags:
  use_new_default_model: true
```

Значение флага в PROD переключают с **false** на **true** — все пользователи, для которых не задана индивидуальная модель, начинают работать с новой моделью.

Пример 2. Флаг, зависящий от пользователя

В таблице пользователей и их групп:

```
user_id | user_group
--------+---------
123     | A
456     | B
```

В конфиге:

```
feature_flags:
  enable_compact_answers: [B]
```

Фича включена только для пользователей группы B, при этом ее можно в любой момент выключить.

Feature Toggles могут быть

- глобальными
- зависящими от пользователя
- DEV toggles

Каждый флаг должен иметь понятную цель и конечный жизненный цикл, иначе конфигурация и код обрастают лишними ветвлениями.

ПРОБЛЕМЫ НАСТРОЕК

Пример. Избыточность параметров

Файл `config.yaml` растет до сотен параметров вида `enable_x`, `enable_y`, `mode_z`, которые никто не трогал годами.

Комбинации значений никто не тестировал, и при случайном изменении одного параметра всплывает баг в редко используемой ветке.

Избыточное количество настроек приводит к конфигурационному взрыву и неконтролируемому числу комбинаций поведения.

Попытка реализовать сложную логику в конфигурации превращает ее в неявный язык программирования без нормальных средств анализа.

ХОРОШИЕ ПРАКТИКИ

Пример 1. Не добавляем все в конфиг

```
1    # ограничение телеграма как платформы
2    MAX_TELEGRAM_MESSAGE_LENGTH = 4000
```

Это не конфигурация, потому что лимит продиктован внешним API и одинаков во всех окружениях.

Пример 2. Документация к параметрам

```
1    # Максимальное число символов вопроса пользователя.
2    # В prod ограничиваем до 600, чтобы не переплачивать за токены.
3    max_prompt_chars: 600
```

Через год по комментарию понятно, почему именно 600, а не 512 или 1000

Конфигурационными должны становиться только те параметры, которые реально нужно менять между окружениями или в эксплуатации.

Важны документация для настроек, разделение инфраструктурных и бизнес-параметров и продуманные значения по умолчанию.

КОНФИГУРАЦИЯ КАК УПРАВЛЯЕМЫЙ СЛОЙ

Пример 1. Перенос между окружениями

Мы хотим воспроизвести баг из prod в test.

Берем PROD-конфиг (без секретов), разворачиваем его в DEV, запускаем ту же версию кода и получаем поведение, близкое к боевому.

Пример 2. Вывод новой фичи в PROD

Добавляем новую модель, включаем ее только в DEV, затем в PROD на 10% пользователей, потом на всех.

Конфигурация – это управляемый слой, определяющий, как код ведет себя в конкретной среде.

Грамотное разделение кода, статических и динамических настроек и Feature Toggles делает систему адаптивной, предсказуемой и сопровождаемой.