Cadence Guide

Что такое Uber Cadence

Uber Cadence — это платформа оркестрации долгоживущих бизнес-процессов.

Cadence используется в ситуациях, когда:

- процесс длится минуты, часы или дни;
- возможны перезапуски, падения сервисов, сетевые проблемы;
- выполнение должно продолжиться **ровно с того места**, где оно остановилось.

Cadence гарантирует:

- сохранность состояния workflow;
- детерминированное выполнение;
- надёжную доставку сигналов.

Основные понятия Cadence

Workflow

Workflow — это **детерминированная функция**, описывающая бизнес-процесс.

Особенности:

- код workflow **не выполняет реальную работу**;
- workflow описывает *последовательность шагов и ожиданий*;
- состояние workflow автоматически сохраняется Cadence.

Пример workflow:

- заказ создан;

- ожидание оплаты;
- подтверждение заказа.

Activity

Activity — это отдельный выполняемый шаг workflow. Activity может быть обычным методом или вызывать внешние системы. Cadence управляет её выполнением, таймаутами и повторными попытками.

Здесь может быть:

- работа с БД;
- HTTP-запрос;
- вызов другого сервиса.

Особенности:

- activity может выполняться повторно (retry);
- activity может падать;
- activity **не детерминирована**, в отличие от workflow.

Workflow Execution

Workflow Execution — это конкретный запущенный экземпляр workflow.

Он идентифицируется:

- workflowId
- (опционально) runId

Позволяет:

- отправлять сигналы в уже запущенный workflow;
- запрашивать его состояние;
- безопасно перезапускать worker.

Worker

Worker — это процесс, который:

- слушает task queue;
- выполняет workflow и activity.

Если worker упал:

- Cadence передаст задачи другому worker;
- состояние workflow сохранится.

Task Queue (Task List)

Task Queue — это очередь задач, которая:

- связывает workflow и worker;
- определяет, какой worker будет исполнять workflow.

В проекте: TASK_QUEUE = "ORDER_TASK_QUEUE"

Worker постоянно опрашивает эту очередь и получает задачи на выполнение.

Signal

Signal — это **асинхронное сообщение** в уже запущенный workflow.

Signal **не запускает новый workflow**, а воздействует на существующий.

Signals и ожидание событий

Пример, когда Workflow может **ждать сигнал**:

- подтверждение оплаты;
- отмена заказа;
- внешний callback.

Используется:

- Workflow.await
- флаг состояния внутри workflow.

Signal:

- не блокирует поток;
- сохраняется в истории;
- доставляется гарантированно.

Query

Query — это способ **прочитать состояние workflow**, не изменяя его.

Особенности:

- query не сохраняется в истории;
- не изменяет состояние;
- всегда возвращает актуальные данные;
- используется для UI, REST API, мониторинга.

Детерминированность Workflow

Workflow **обязан быть детерминированным**.

Это значит:

- нельзя использовать new Date();
- нельзя использовать Thread.sleep;
- нельзя обращаться к внешнему миру напрямую.

Почему? Cadence может **перепроигрывать workflow** (replay), и результат должен быть идентичен.

Разрешено:

- Workflow.sleep

- Workflow.await
- Duration
- сигналы и queries

Saga (компенсации) — ключевой паттерн Cadence

Что такое Saga

Saga — это паттерн управления распределённой транзакцией через компенсации.

Вместо классического rollback:

- каждый шаг имеет **компенсирующее действие**;
- при ошибке выполняются компенсации в обратном порядке.

Зачем нужна Saga

Пример бизнес-процесса:

1. Создать заказ
2. Списать деньги
3. Зарезервировать товар

Если шаг 3 упал:

- заказ уже создан;
- деньги списаны;

- транзакцию БД откатить нельзя.

Saga решает эту проблему.

Saga в Cadence

Cadence предоставляет класс: com.uber.cadence.workflow.Saga

Он позволяет:

- регистрировать компенсации;
- автоматически запускать их при ошибке;
- гарантировать порядок выполнения (LIFO).

Принцип работы Saga

1. Выполняется шаг
2. Сразу регистрируется компенсация
3. Выполняется следующий шаг
4. При ошибке:

- запускаются компенсации в обратном порядке

Важно:

- компенсации — это бизнес-логика;
- они тоже могут падать;
- они должны быть идемпотентными.

Пример Saga в Order Workflow (концептуально)

- Создали заказ → компенсация: отменить заказ
- Подтвердили оплату → компенсация: вернуть деньги

- Завершили заказ → компенсации больше не нужны

Пример кода Saga (упрощённо):

```
Saga saga = new Saga(new Saga.Options.Builder()
        .setParallelCompensation(false)
        .build());

try {
    createOrder();
    saga.addCompensation(this::cancelOrder);

    waitForPayment();
    saga.addCompensation(this::refundPayment);

    completeOrder();
} catch (Exception e) {
    saga.compensate();
    throw e;
}
```

Два способа делать Saga в Cadence

1. Через com.uber.cadence.workflow.Saga (канонический способ)

```
Saga saga = new Saga(new Saga.Options.Builder().build());


try {
    activities.createOrder();
    saga.addCompensation(activities::cancelOrder);

    activities.chargePayment();
    saga.addCompensation(activities::refundPayment);

    activities.reserveProduct();
    saga.addCompensation(activities::releaseProduct);
} catch (Exception e) {
    saga.compensate();
    throw e;
}
```

Что делает saga.compensate()

- вызывает **все зарегистрированные компенсации**
- в **обратном порядке**
- как activity
- с гарантией Cadence (replay, retries)

2. Ручная Saga через activity

```
try {
    createOrderActivity();
    chargePaymentActivity();
    reserveProductActivity();
} catch (Exception e) {
    refundPaymentActivity();
    cancelOrderActivity();
}
```

Компенсации:

- вызываются явно
- логика отката — твоя разработчика
- Cadence просто исполняет activity

Как работает Order Demo

Шаг 1. Создание заказа

REST-запрос:

POST /order/{orderId}

Что происходит:

- создаётся workflow;
- статус становится PAYMENT_PENDING.

Шаг 2. Подтверждение оплаты

REST-запрос:

POST /order/{orderId}/paid

Что происходит:

- отправляется signal в workflow;
- workflow продолжает выполнение.

Шаг 3. Получение статуса

REST-запрос:

GET /order/{orderId}/status

Возвращает:

- текущее состояние workflow.

Коротко о том, что происходит при выполнении:

1. Клиент стартует workflow execution
2. Cadence сохраняет событие WorkflowStarted
3. Worker:
    - берёт task из task queue
    - исполняет workflow код
4. Worker падает □
5. Cadence:

    - ничего не теряет
    - ждёт нового worker
6. Новый worker:
    - replay’ит историю
    - продолжает выполнение

Полный жизненный цикл выполнения Cadence

(с Workflow, Activity, Worker, Task Queue, History, Signal, Query, Saga)

- **Client** — стартует workflow, шлёт signals, делает queries
- **Cadence Server** — хранит состояние и историю
- **Workflow Execution** — конкретный экземпляр бизнес-процесса
- **Worker** — исполняет код
- **Task Queue (Task List)** — очередь задач
- **Workflow Task** — задача на исполнение workflow-кода
- **Activity Task** — задача на исполнение activity
- **Activity** — недетерминированный, внешний код
- **History** — журнал событий workflow

Шаг за шагом

1. Клиент стартует workflow execution

WorkflowClient.start(workflow::processOrder, "ORDER-1");

Что делает Cadence:

- создаёт **Workflow Execution**
- присваивает workflowId

- пишет событие в history:

WorkflowExecutionStarted

 Ни один worker ещё ничего не выполнял

2. Cadence создаёт Workflow Task

- кладёт **Workflow Task** в **Workflow Task Queue**
- task содержит:
    - ссылку на workflow execution
    - указание: «нужно выполнить workflow-код»

3. Worker забирает Workflow Task

Worker:

- подписан на **task queue**
- получает **Workflow Task**
- загружает **полную history** workflow

4. Worker исполняет Workflow (replay + execution)

Worker:

1. **Replay**:
    - проигрывает всю историю
    - восстанавливает состояние workflow

2. **Execution**:
    - выполняет код workflow **до первого блокирующего места**

Пример:

```
createOrder();
Workflow.await(paymentConfirmed);
```

5. Workflow вызывает Activity

В workflow-коде:

```
activities.reserveProduct();
```

Что происходит на самом деле:

Cadence:

- пишет событие в history:

ActivityTaskScheduled

- кладёт **Activity Task** в **Activity Task Queue**

ВАЖНО

- workflow **НЕ выполняет activity**
- workflow **ставит activity в очередь**

6. Worker (или другой worker) исполняет Activity

Worker:

- забирает **Activity Task**
- выполняет обычный Java-код:
    - HTTP
    - DB
    - Kafka
    - REST

После выполнения:

- Cadence пишет в history: ActivityTaskCompleted или ActivityTaskFailed

7. Cadence создаёт новый Workflow Task

После завершения activity:

- Cadence создаёт **новый Workflow Task**
- кладёт его в workflow task queue

8. Worker продолжает Workflow

Worker:

- replay’ит историю
- видит, что activity уже завершена
- продолжает выполнение workflow-кода

9. Signal (асинхронное событие)

Когда приходит signal: workflow.paymentConfirmed();

Cadence:

- пишет в history:

WorkflowExecutionSignaled

- создаёт **Workflow Task**

Worker:

- replay

- видит, что paymentConfirmed = true
- Workflow.await() разблокируется

□ Что если worker падает?

□ Worker падает в любой момент

Cadence:

- ничего не теряет
- history сохранена

□ Новый worker стартует

- получает Workflow Task
- **replay’ит history**
- восстанавливает состояние
- продолжает выполнение

☞ Activity **не переисполняются**, если они уже завершены

□ Где здесь Saga?

Saga — часть **workflow-кода**:

- регистрация компенсаций — детерминированная
- сами компенсации — activity

При ошибке:

- workflow вызывает saga.compensate()
- Cadence создаёт activity tasks для компенсаций