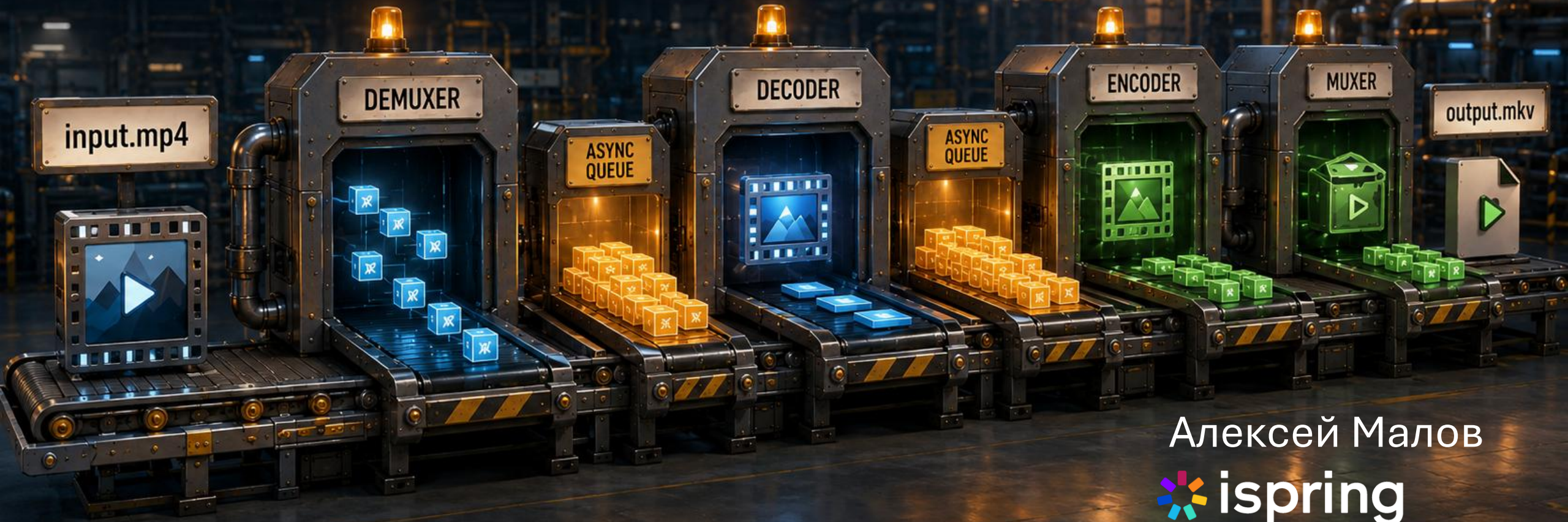
FFmpeg без боли:

архитектура видео-транскодера на C++23

C++23 RAII modules expected async pipeline backpressure

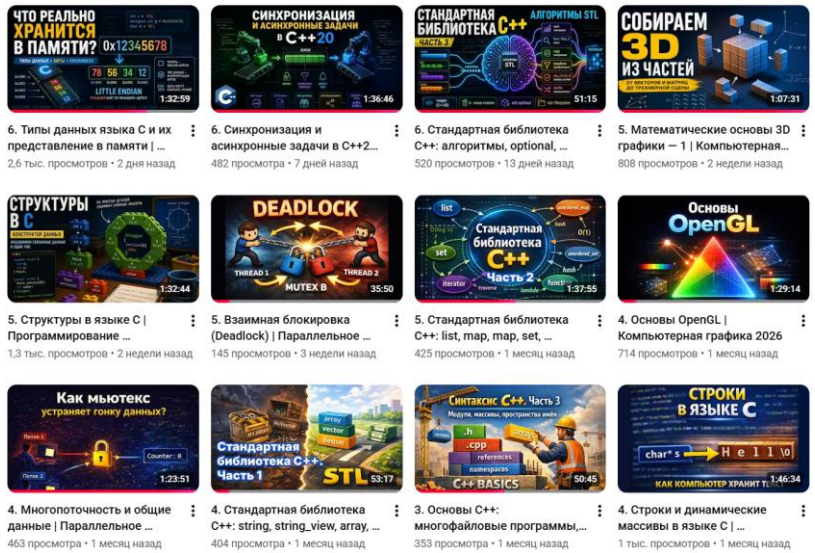


Алексей Малов

ispring

Обо мне

- Ведущий архитектор в iSpring
- Преподаю компьютерные дисциплины
- Веду образовательный канал на YouTube
  - https://youtube.com/@vividbw



ispring

iSpring Suite – конструктор электронных курсов на базе PowerPoint

Нужно ускорить транскодирование видео

- iSpring использует библиотеку FFmpeg для обработки видео
- Возникла задача ускорить обработку видео
- Использовать аппаратное ускорение
  - Intel QSV
  - NVIDIA
  - …

Надо ускорить транскодинг видео.

Приключение на пару недель

Кадр из анимационного сериала «Рик и Морти» © Adult Swim.

Проблема №1 – обновить FFmpeg

- Библиотека мультимедиа не обновлялась с 2018 года
  - FFmpeg 4
  - Актуальная версия: 7.1
  - API сильно поменялся
- Переписали почти всю библиотеку
- 💡 Лучше обновляться по-маленькому, а не по большому

Проблема №2 – немасштабируемая архитектура транскодера

- Аппаратное ускорение не помогло так, как ожидалось
- Причина: последовательный транскодер
  - CPU недогружен
- Решение:
  - Распараллеливание
  - Новые интересные задачи
- Итог: ускорили транскодирование видео на 30-60%

О чем мой доклад

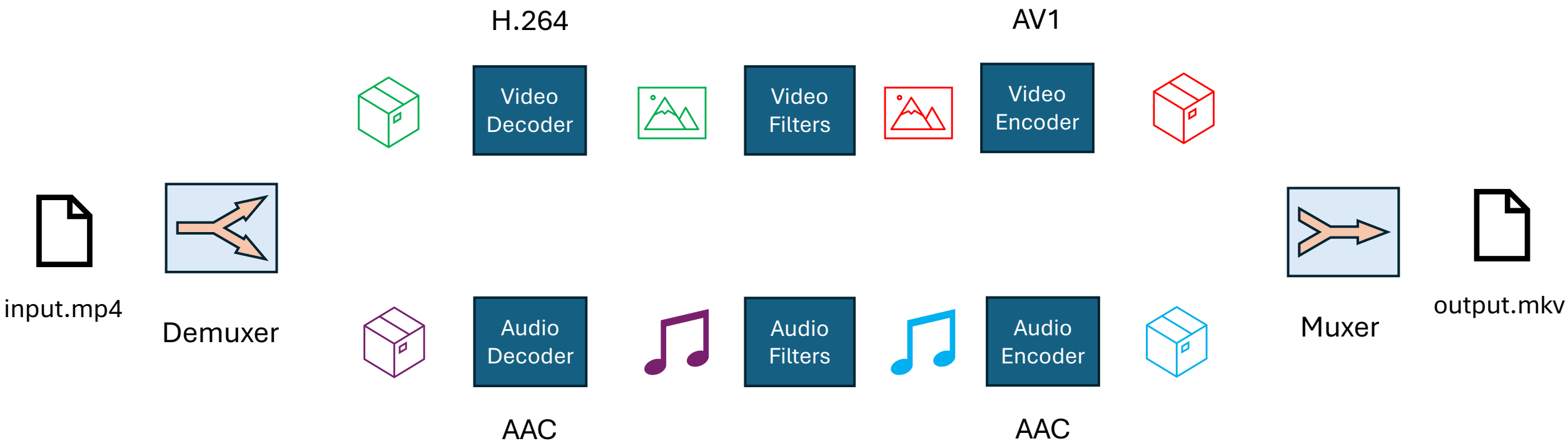
- Минимум про устройство FFmpeg
- Безопасная интеграция C API в C++23 (modules + RAII + expected)
- Архитектура транскодера: от последовательного к асинхронному
- Очереди, backpressure и pipeline
- Разбор производительности и ограничения

Кратко об FFmpeg

Что такое FFmpeg

- Ffmpeg – мультимедийный фреймворк
- Позволяет:
  - читать и писать видео- и аудиофайлы
    - demux (разбор контейнера на потоки)
    - mux (сборка обратно)
  - кодировать и декодировать
  - конвертировать форматы
  - обрабатывать и фильтровать аудио и видео
- Поддерживает:
  - практически все форматы (от устаревших до современных)
- Особенности:
  - кроссплатформенный с открытым исходным кодом
  - де-факто стандарт индустрии

Как FFmpeg обрабатывает видео



FFmpeg C API: ручное управление ресурсами

```
AVFormatContext *inputContext = nullptr;
int ec = avformat_open_input(&inputContext, "video.mp4", nullptr, nullptr); // Открываем файл
if (ec < 0) {
    return Error("Failed to input video");
}

AVPacket *packet = av_packet_alloc(); // Аллоцируем пакет
if (!packet) {
    return Error("allocation failed");
}

ec = av_read_frame(inputContext, packet); // Читаем пакет из файла
if (ec < 0) {
    return Error("read failed");
}

// работа с packet ...

av_packet_free(&packet); // Освобождаем пакет
avformat_close_input(&inputContext); // Закрываем файл
```

FFmpeg C API: ручное управление ресурсами

```
AVFormatContext *inputContext = nullptr;
int ec = avformat_open_input(&inputContext, "video.mp4", nullptr, nullptr); // Открываем файл
if (ec < 0) {
    return Error("Failed to input video");
}

AVPacket *packet = av_packet_alloc(); // Аллоцируем пакет
if (!packet) {
    avformat_close_input(&inputContext);  //  !
    return Error("allocation failed");
}

ec = av_read_frame(inputContext, packet); // Читаем пакет из файла
if (ec < 0) {
    av_packet_free(&packet);              //  !
    avformat_close_input(&inputContext);  //  !
    return Error("read failed");
}

// работа с packet ...

av_packet_free(&packet); // Освобождаем пакет
avformat_close_input(&inputContext); // Закрываем файл
```

Безопасная интеграция с C API

Цели при проектировании API

- Современный C++ 23
  - C++ modules вместо #include
- Безопасность
  - RAII для управления ресурсами
- Производительность
  - “тонкие” обёртки
- Изоляция FFmpeg
  - скрыть C API за модулем
  - не распространять детали реализации

Что такое C++ 20 modules

- Модуль = Интерфейс + Реализация

```
// module.ixx – interface unit
export module my_module;

class NonExportedClass { … };

export class ExportedClass {
public:
    ExportedClass();
private:
    NonExportedClass m_packet;
};
```

```
// module.cpp - implementation unit
module my_module;

NonExportedClass::NonExportedClass () {
    …
}

ExportedClass::ExportedClass() {
    ...
}
```

Что такое C++ 20 modules

- Модуль = Интерфейс + Реализация

```
// module.ixx – interface unit
module;

#include <some_lib.hpp>

export module my_module;

class NonExportedClass { … };

export class ExportedClass {
public:
    ExportedClass();
private:
    NonExportedClass m_packet;
};
```

Global module fragment

```
// module.cpp - implementation unit
module;

#include "some_other_lib.hpp"

module my_module;

NonExportedClass::NonExportedClass () {
    …
}

ExportedClass::ExportedClass() {
    ...
}
```

Global module fragment

Bridge header для C-библиотеки

- FFmpeg — C API
  - Заголовки не C++ friendly
  - Интеграция с C++ - забота интегратора, а не библиотеки
- Решение: extern "C"
  - ```
    extern "C" {
    #include <libavcodec/packet.h>
    }
    ```
- Проблема:
  - в global module fragment могут быть только директивы препроцессора
- Решение – bridge header

```
// packet.hpp
#pragma once

extern "C" {
#include <libavcodec/packet.h>
}
```

```
// packet.ixx
module;

#include "packet.hpp"

export module ffmpeg.packet;
```
```

Класс Packet

```cpp
// packet.ixx
module;

#include "../src/packet.hpp"
#include <cassert>

export module ffmpeg.packet;
import std;

namespace ffmpeg {

export class Packet {
public:
    Packet(); // Создаёт пустой пакет (без данных)
    static Packet Null() noexcept { return Packet{ nullptr }; } // Создаёт Null Packet (используется как EOF-маркер)
    …
private:
    explicit Packet(std::nullptr_t) noexcept : m_packet{ nullptr } { }

    struct Deleter {
        void operator()(AVPacket* packet) const noexcept { av_packet_free(&packet); }
    };

    std::unique_ptr<AVPacket, Deleter> m_packet;
};

} // namespace ffmpeg
```

```cpp
// packet.hpp
#pragma once

extern "C" {
#include <libavcodec/packet.h>
}
```

Класс Packet (продолжение)

```cpp
export class Packet {
public:
    Packet();

    [[nodiscard]] explicit operator bool() const noexcept {
        return m_packet.get() != nullptr;
    }

    template <typename Self> // C++ 23 «deducing this»
    [[nodiscard]] auto get(this Self& self) noexcept -> std::conditional_t<std::is_const_v<Self>, const AVPacket*, AVPacket*> {
        return self.m_packet.get();
    }

    template <typename Self>
    [[nodiscard]] decltype(auto) operator*(this Self& self) noexcept {
        assert(self.get() != nullptr);
        return *self.get();
    }

    template <typename Self>
    [[nodiscard]] auto operator->(this Self& self) noexcept {
        assert(self.get() != nullptr);
        return self.get();
    }
private:
    std::unique_ptr<AVPacket, Deleter> m_packet;
};
```

```cpp
[[nodiscard]] const AVPacket* get() const noexcept {
    return self.m_packet.get();
}

[[nodiscard]] AVPacket* get() noexcept {
    return self.m_packet.get();
}
```

Файл реализации модуля ffmpeg.packet

```
// packet.cpp
module;
#include "packet.hpp"
#include "error.hpp"
#include <cerrno>

module ffmpeg.packet;
import ffmpeg.error;
import std;

namespace ffmpeg {

Packet::Packet()
    : m_packet{ av_packet_alloc() }
{
    if (!m_packet) [[unlikely]] {
        ThrowFFmpegNoMem("av_packet_alloc");
    }
}

} // namespace ffmpeg
```

Стратегия обработки ошибок

- Два механизма
  - std::expected для Try*-методов
  - исключения — только в конструкторах
- Почему так
  - ошибка должна быть явно обработана
  - [[nodiscard]] защищает от игнорирования
  - нет исключений в обычном потоке управления
  - нельзя получить некорректно сконструированный объект
- Дополнительно
  - в некоторых классах есть throwing-обёртки над Try*
  - можно выбрать стиль: expected или exceptions

Хранение информации об ошибках

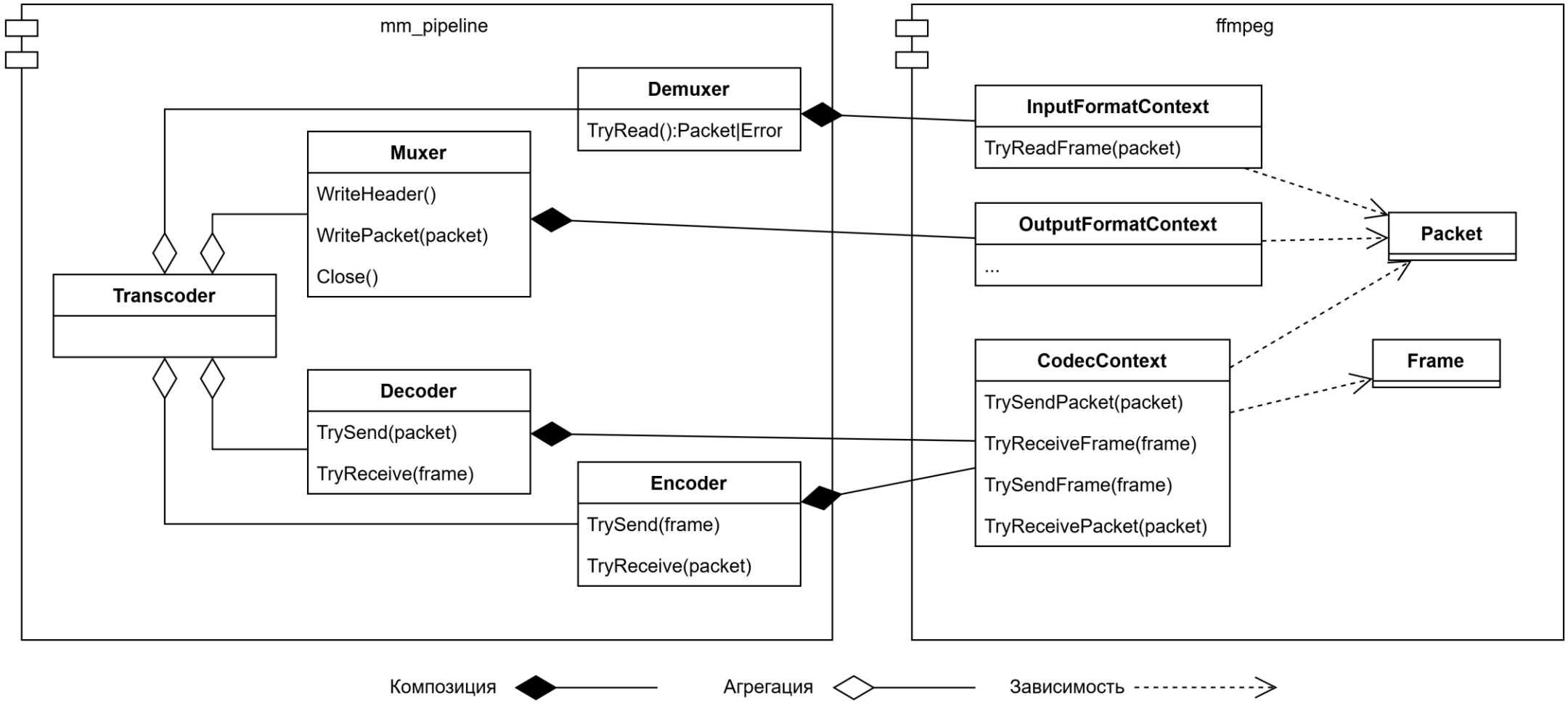
```
namespace ffmpeg {

export struct Error final {
    int code = 0;
    const char* where = nullptr;
    constexpr bool Ok() const noexcept { return code == 0; }
    std::error_code ToErrorCode() const noexcept;
};

export inline Error MakeError(int code, const char* where) {
    return { .code = code, .where = where };
}

export [[nodiscard]] inline std::unexpected<Error> MakeUnexpected(int code, const char* where) noexcept {
    return std::unexpected<Error>(std::in_place, code, where);
}

export [[nodiscard]] inline std::expected<void, Error> ExpectedFromErrorCode(int code, const char* where) noexcept {
    if (code < 0) [[unlikely]] { return MakeUnexpected(code, where); }
    return {};
}

} // namespace ffmpeg
```

Смешанный подход к возврату ошибок

```
// Возвращает ссылку на exp, либо бросает исключение
export template <class Expected> decltype(auto) Check(Expected&& exp);

std::expected<void, Error> OutputFormatContext::TryWritePacket(AVPacket& packet) noexcept {
    assert(m_ctx);
    assert(!m_state.closed && "OutputFormatContext::TryWritePacket(closed)");
    assert(m_state.headerWritten && "TryWritePacket called before header");

    return ExpectedFromErrorCode(
        av_interleaved_write_frame(m_ctx.get(), &packet), "av_interleaved_write_frame");
}

void OutputFormatContext::WritePacket(AVPacket& packet) {
    Check(TryWritePacket(packet));
}
```

Последовательный pipeline и его пределы

Архитектура транскодера



Устройство транскодера

```
export class Transcoder {
public:
    void Run() { … }

private:
    struct Branch {
        Decoder& decoder;
        Encoder& encoder;
        int streamIndex = -1;
        Muxer::TrackId track;
    };

    Muxer& m_muxer;
    Demuxer& m_demuxer;
    Branch m_video;
    Branch m_audio;
};
```

Транскодер: главный цикл

```
using FrameProcessor = std::function<void(Frame&)>;

void Run() {
    std::int64_t videoFrameCounter = 0;
    const FrameProcessor processVideo = [&videoFrameCounter](Frame& frame) {
        frame->pts = videoFrameCounter++;
    };

    std::int64_t audioPtsSamples = 0;
    const FrameProcessor processAudio = [&audioPtsSamples](Frame& frame) {
        frame->pts = audioPtsSamples;
        audioPtsSamples += frame->nb_samples;
    };

    for (Packet pkt; pkt = m_demuxer.Read();) {
        if (pkt->stream_index == m_video.streamIndex) SendPacket(pkt, m_video, processVideo);
        if (pkt->stream_index == m_audio.streamIndex) SendPacket(pkt, m_audio, processAudio);
    }

    Flush(m_video, processVideo);
    Flush(m_audio, processAudio);
}
```

Цикл работы с декодером

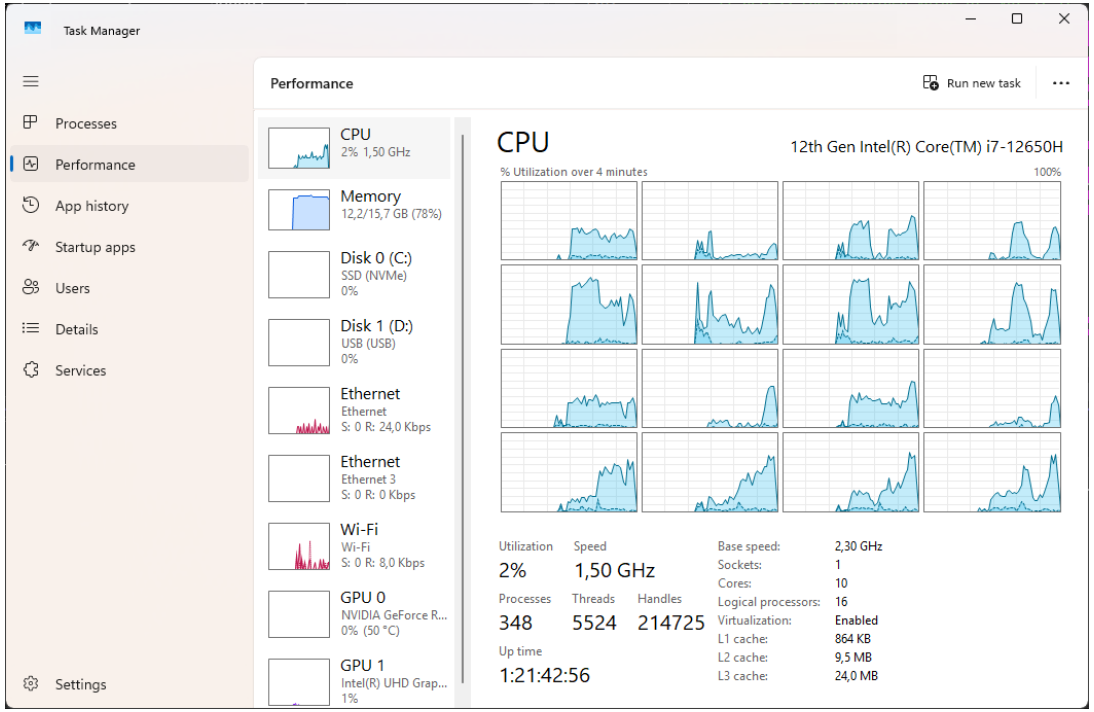
```
void SendPacket(const Packet& packet, Branch& branch, const FrameProcessor& processor) {
    while (branch.decoder.Send(packet) != ffmpeg::SendResult::Accepted) {
        DrainDecoder(branch, processor);
    }
    DrainDecoder(branch, processor);
}

void DrainDecoder(Branch& branch, const FrameProcessor& process) {
    Frame frame;
    while (branch.decoder.Receive(frame) == ffmpeg::ReceiveResult::Produced) {
        process(frame);

        SendFrame(frame, branch);
    }
}
```

Цикл работы с энкодером и завершение работы

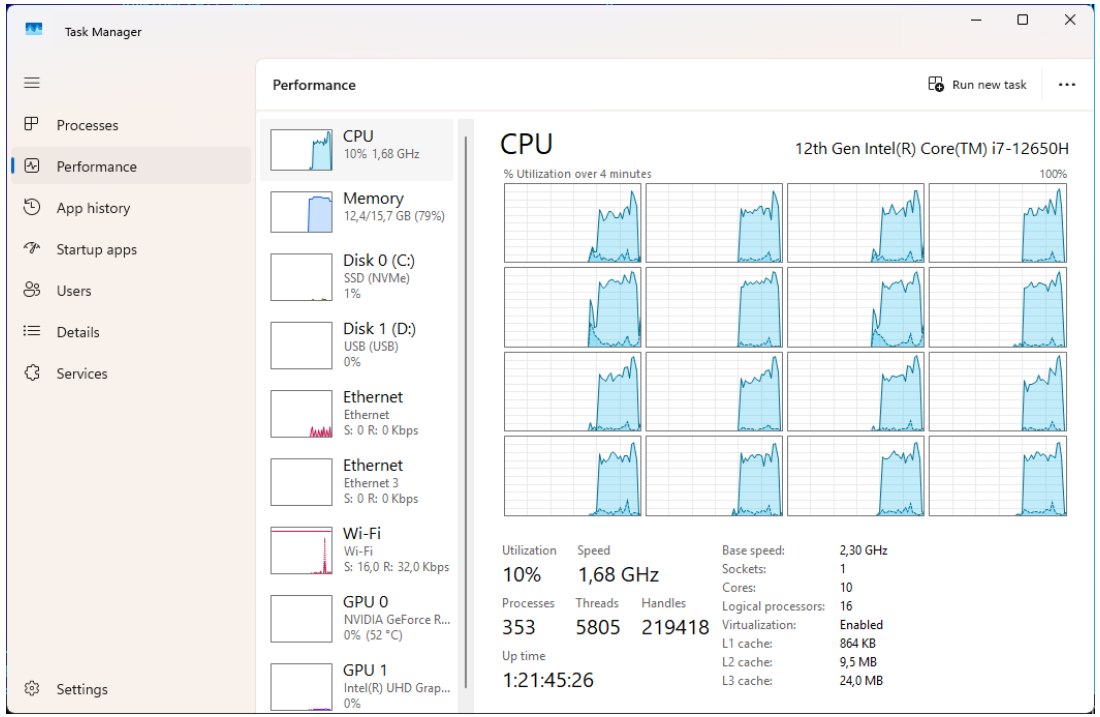
```
void SendFrame(const Frame& frame, Branch& branch) {
    while (branch.encoder.Send(frame) == ffmpeg::SendResult::NeedReceive) {
        DrainEncoder(branch);
    }
    DrainEncoder(branch);
}

void DrainEncoder(Branch& branch) {
    Packet pkt;
    while (branch.encoder.Receive(pkt) == ffmpeg::ReceiveResult::Produced) {
        m_muxer.WritePacket(branch.track, *pkt, branch.encoder.GetStreamTimeBase().ToAV());
    }
}

void Flush(Branch& branch, const FrameProcessor& process) {
    SendPacket(Packet::Null(), branch, process);
    SendFrame(Frame::Null(), branch);
}
```

Оценим производительность транскодера

Наш транскодер: 108.14c



ffmpeg.exe – 64.38c

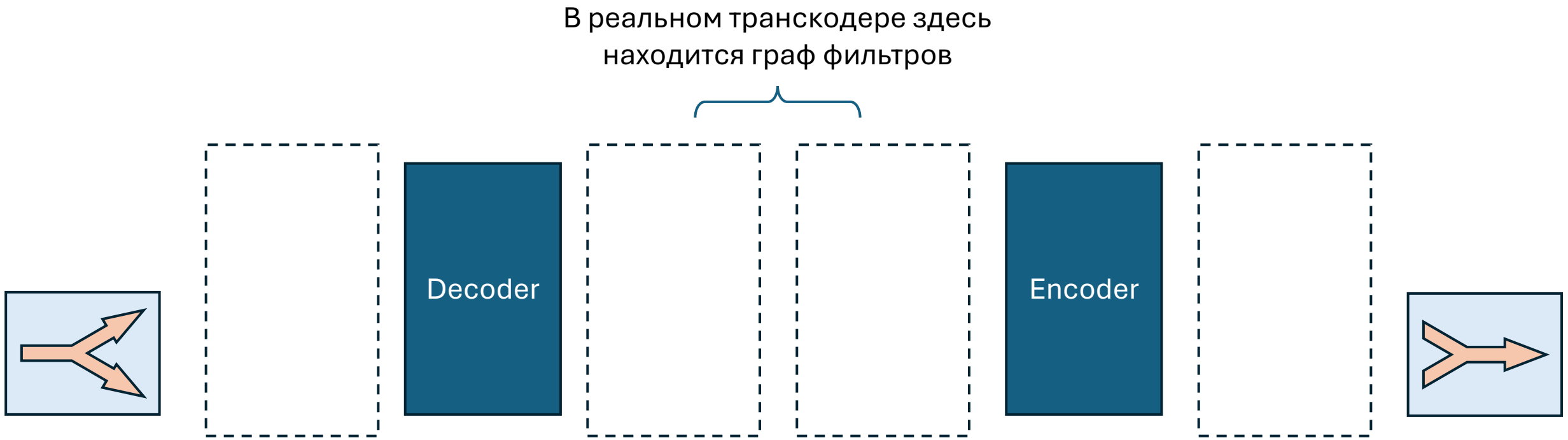


Переход к асинхронности

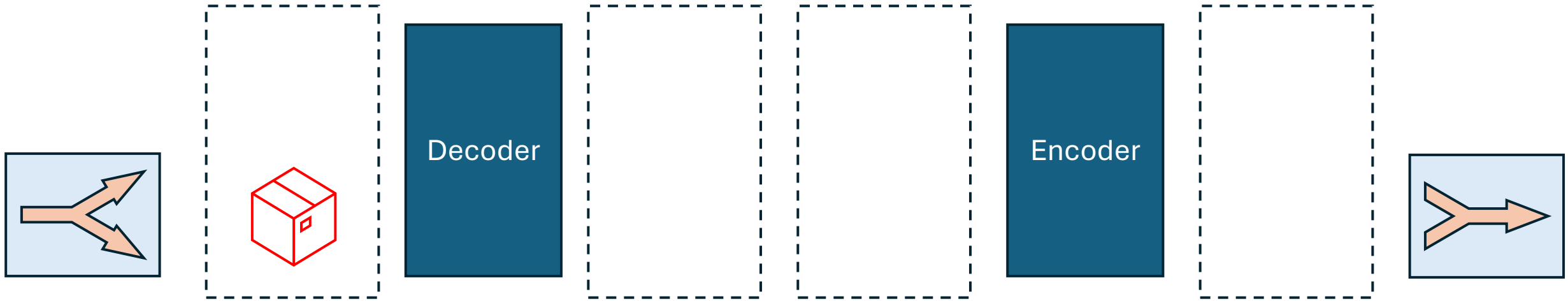
Переход к параллельному конвейеру

- Decoder и Encoder должны работать в разных потоках
  - [queue] → Packet → Decoder → Frame → [queue]
  - [queue] → Frame → Encoder → Packet → [queue]
- Проблема: decoder быстрее encoder
  - Риск: очередь фреймов может бесконтрольно расти
- Решение: bounded queue между стадиями
  - Producer: decoder помещает Frame в очередь
  - Consumer: encoder извлекает Frame из очереди
- Backpressure: если очередь заполнена, producer ждёт

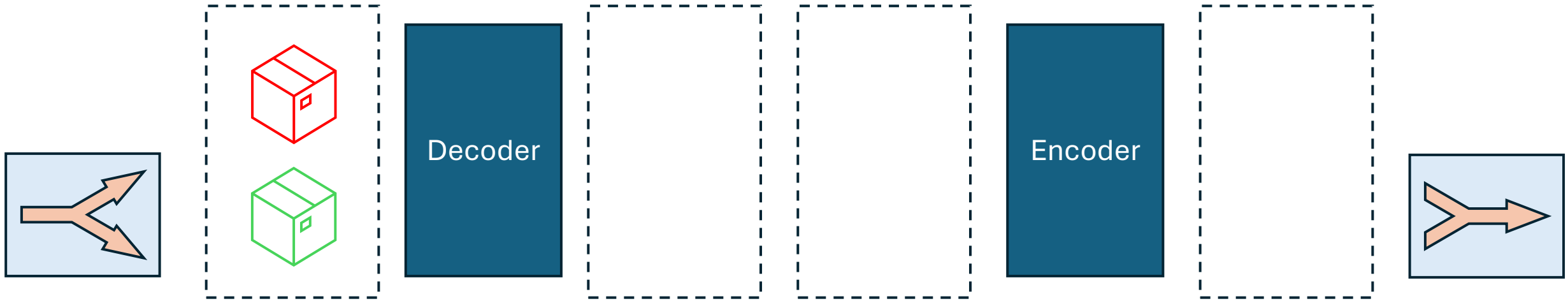
Параллельный конвейер



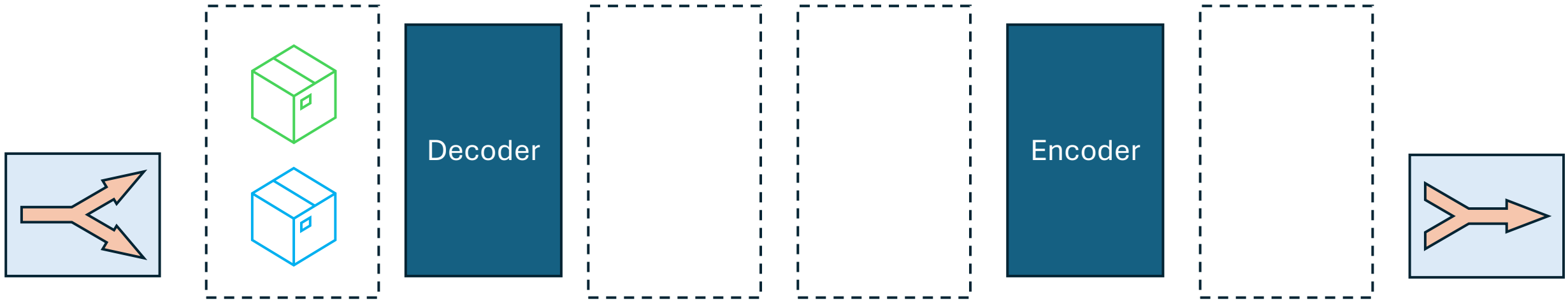
Параллельный конвейер



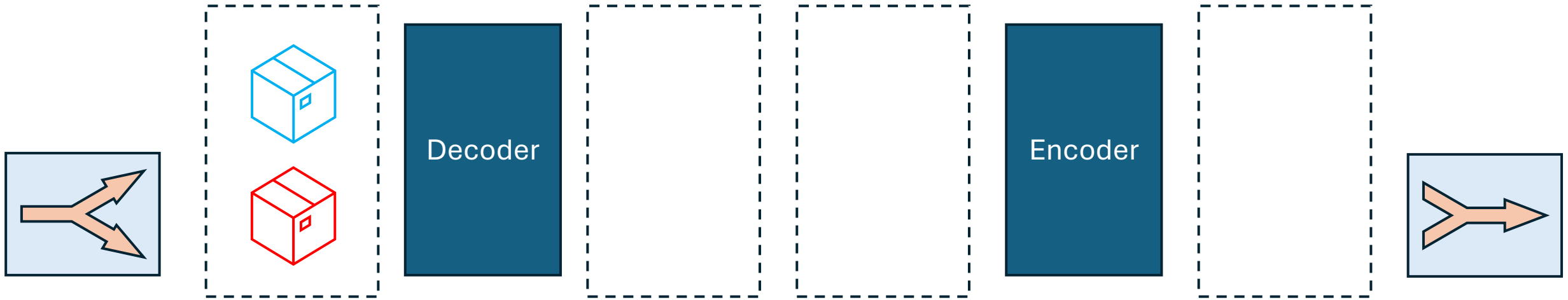
Параллельный конвейер



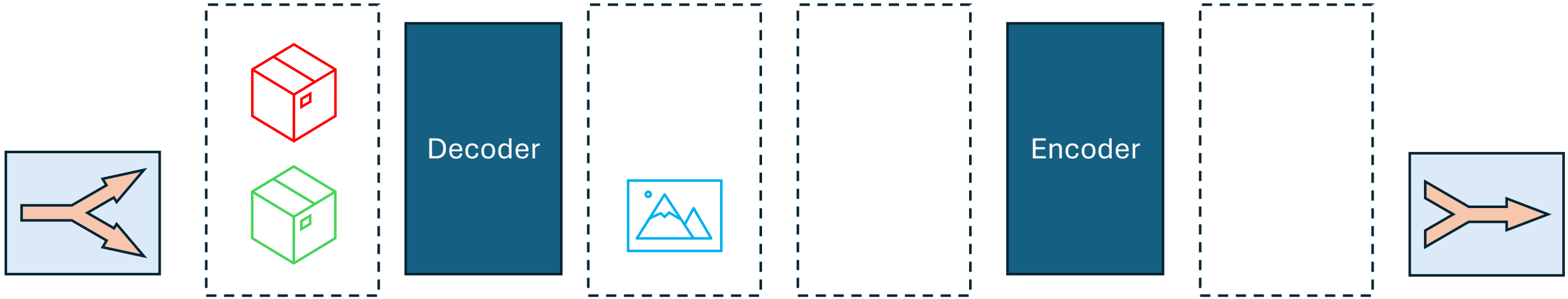
Параллельный конвейер



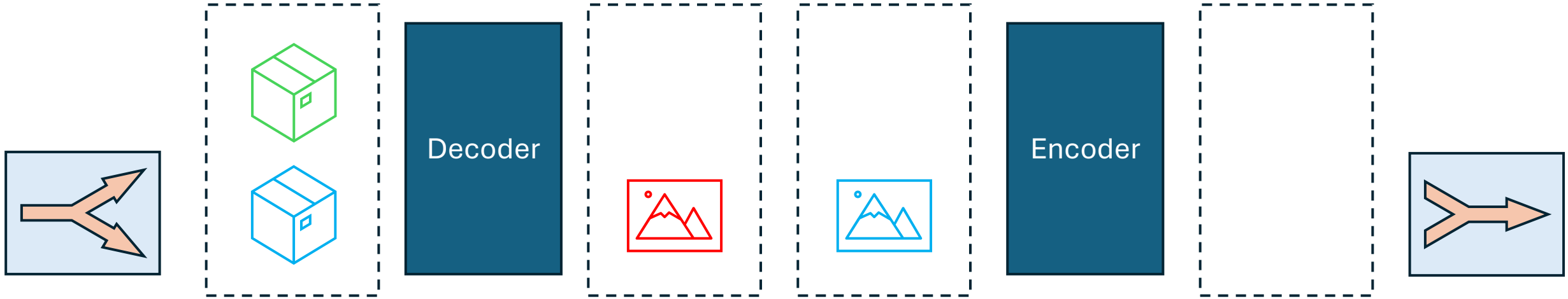
Параллельный конвейер



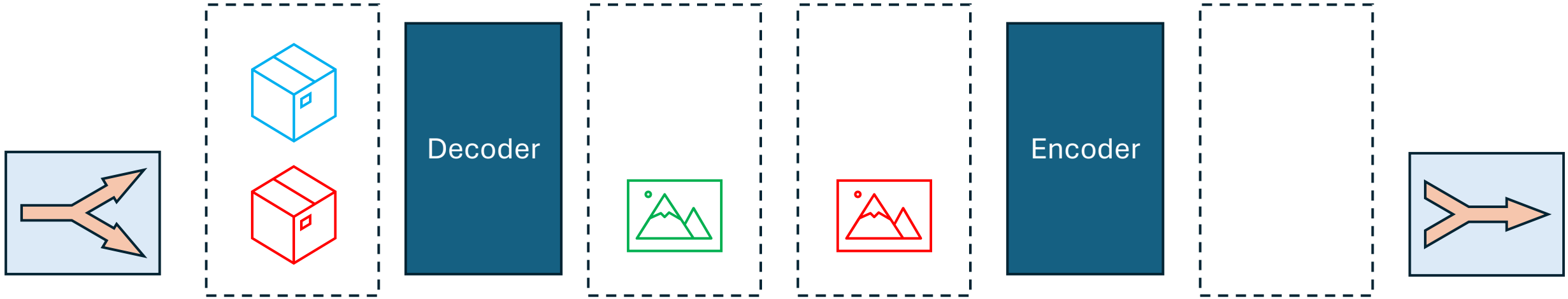
Параллельный конвейер



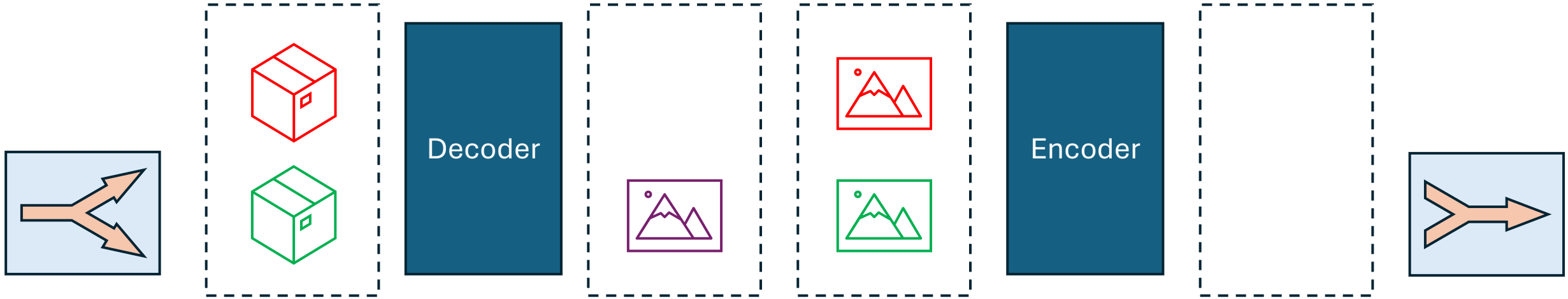
Параллельный конвейер



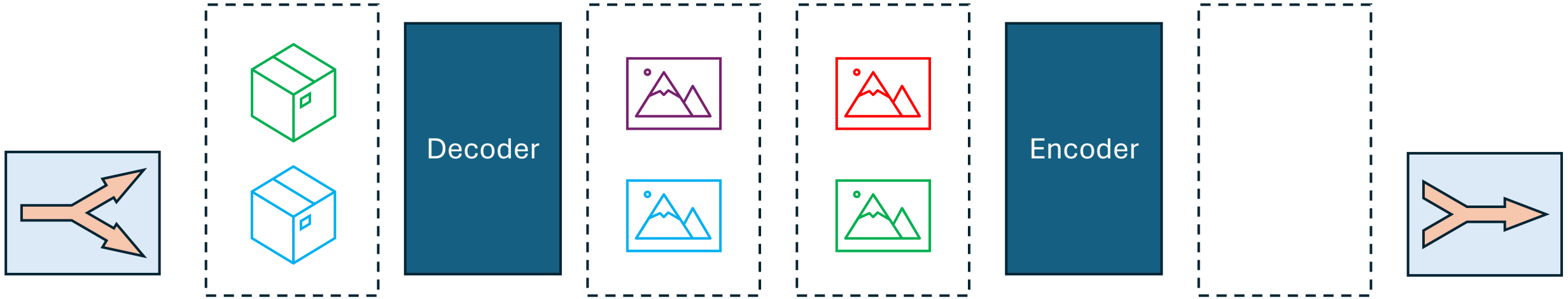
Параллельный конвейер



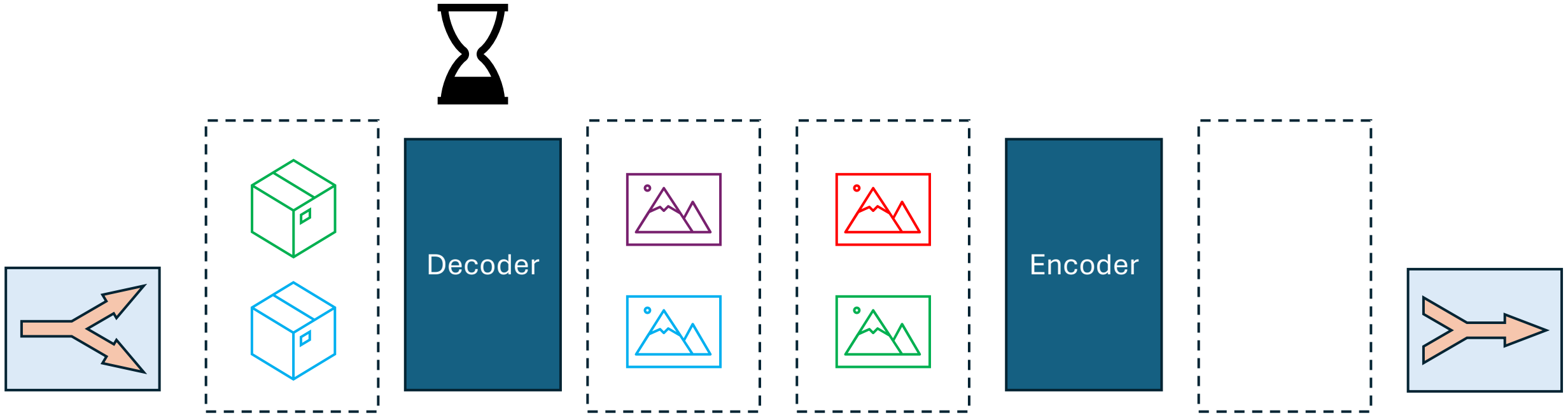
Параллельный конвейер



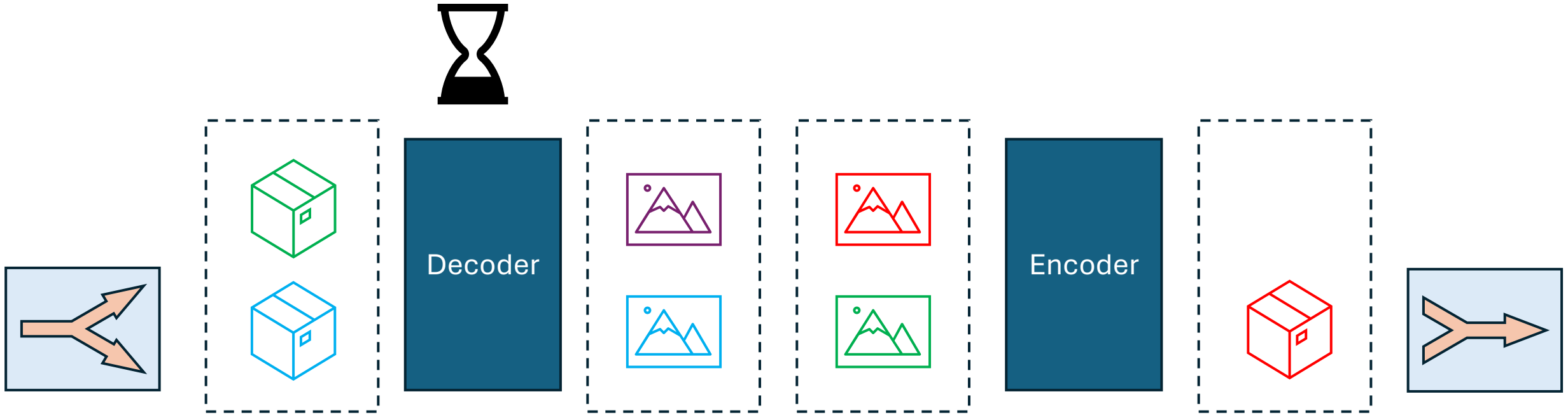
Параллельный конвейер



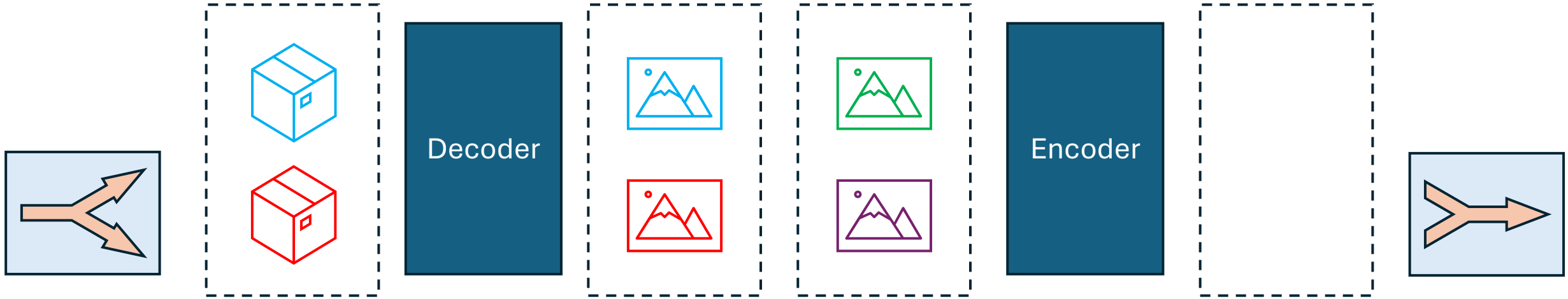
Параллельный конвейер



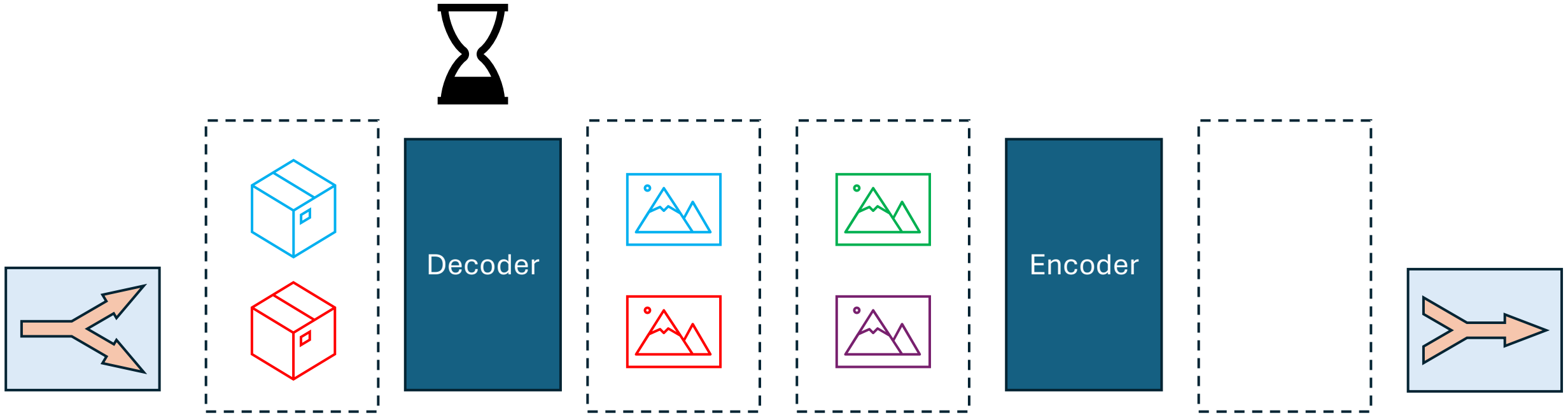
Параллельный конвейер



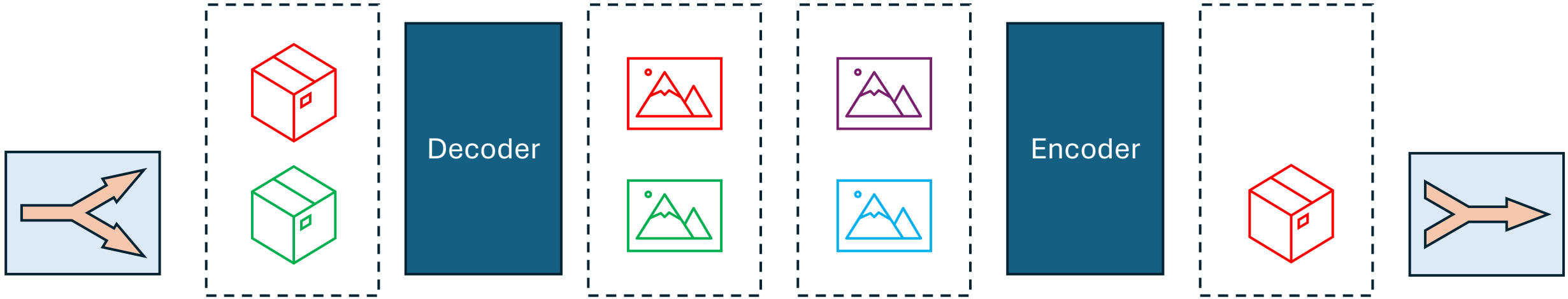
Параллельный конвейер



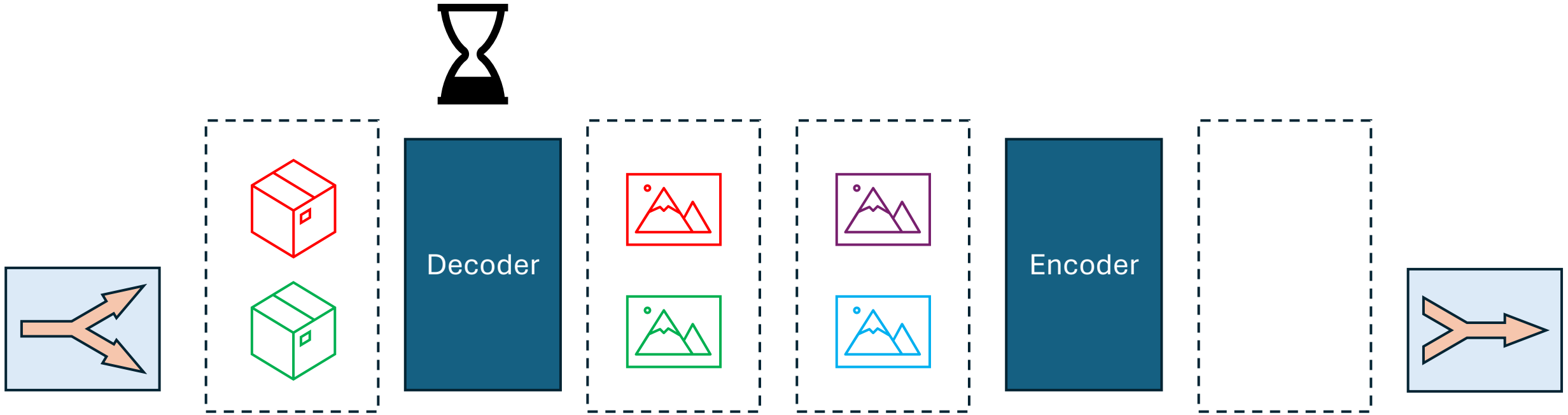
Параллельный конвейер



Параллельный конвейер

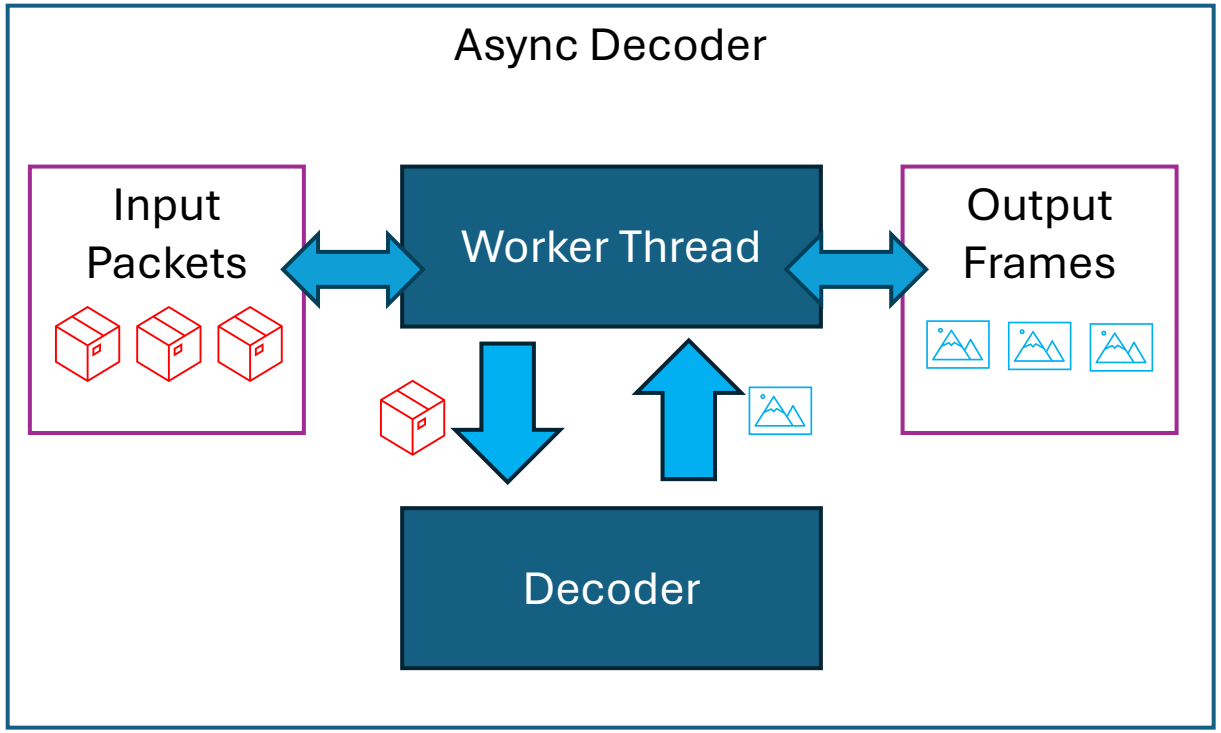


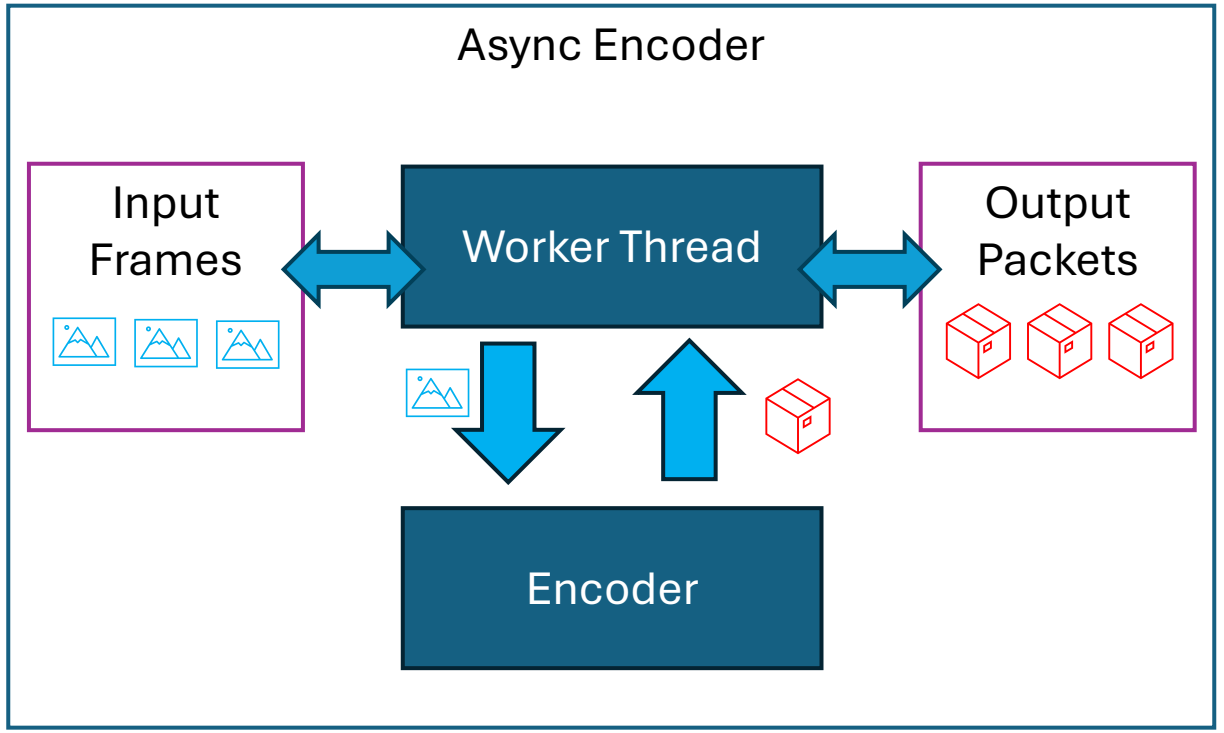
Параллельный конвейер



Компоненты асинхронного конвейера

Асинхронные encoder и decoder





Cooperative cancellation в C++20

- stop_source – отправляет сигнал остановки
  - stop_source ss;
    ss.request_stop();
- stop_token – позволяет получить сигнал остановки
  - stop_token st = ss.get_token();
    ...
    if (st.stop_requested()) { ... }

```
void Worker(std::stop_token st) {
    while (!st.stop_requested())
        DoWork();
}

int main() {
    std::jthread thread{ Worker };
    ...
    thread.request_stop();
}
```

```
std::condition_variable_any cv;

void CancelableWait(std::stop_token st) {
    std::unique_lock lock{ m_mutex };

    // wait вернёт управление, если
    // через stop_token придёт сигнал stop
    v.wait(lock, st, [] { … });
}
```

Асинхронная очередь

```cpp
export template <typename T>
class AsyncQueue : private detail::AsyncQueueBase {
    std::deque<T> m_queue;
public:
    enum class TryPushResult { Ok, Full, Closed };
    enum class TryPopError { Empty, Closed };

    explicit AsyncQueue(size_t capacity);

    TryPushResult TryPush(const T& value, bool ignoreCapacity = false);
    TryPushResult TryPush(T&& value, bool ignoreCapacity = false);
    std::expected<T, TryPopError> TryPop();

    void PushOrWait(T value, std::stop_token st);
    T PopOrWait(std::stop_token st);
    void PopAllOrWait(std::deque<T>& out, std::stop_token st);

    size_t TryPopAll(std::deque<T>& out);
    size_t PopUpToOrWait(std::deque<T>& out, size_t maxItems, std::stop_token st);

    void Close();
    [[nodiscard]] bool IsClosed() const noexcept;
    void NotifyAll() noexcept;
};
```

```cpp
namespace detail {
class AsyncQueueBase {
protected:
    ...
    const size_t m_capacity;
    mutable std::mutex m_mutex;
    std::condition_variable_any m_cvNotEmpty;
    std::condition_variable_any m_cvNotFull;
    bool m_closed = false;
};
} // namespace detail
```

Неблокирующие вызовы

Ожидающие вызовы

Batching-методы

Жизненный цикл

Интерфейс класса AsyncStage

```
export template <class Processor, class Input, class Output>
class AsyncStage : private detail::AsyncStageBase {
public:
    AsyncStage(Processor processor, size_t inputCapacity, size_t outputCapacity, PipelineNotifier& notifier);

    void Start();
    void RequestStop();

    bool TryPush(Input&& input);
    void CloseInput();
    std::optional<Output> TryPop();
    …
private:
    using InputQueue = AsyncQueue<Input>;
    using OutputQueue = AsyncQueue<Output>;

    InputQueue m_inputQueue;
    OutputQueue m_outputQueue;
    Processor m_processor;
    bool m_inputClosed = false;
};
```

```
namespace detail {
class AsyncStageBase {
protected:
  enum class State { NotStarted, Running, Finished,
                     Stopped, Failed, };
  ...
  PipelineNotifier& m_pipelineNotifier;
  std::exception_ptr m_workerException;
  std::atomic<State> m_state = State::NotStarted;
  std::jthread m_workerThread;
};
} // namespace detail
```

Рабочий цикл AsyncStage

```
void WorkerThreadFunc(std::stop_token stopToken) override {
    try {
        std::deque<Input> batch;

        while (true) {
            m_inputQueue.PopAllOrWait(batch, stopToken);
            m_pipelineNotifier.Notify();

            for (Input& input : batch) {
                const bool eof = !input;

                ProcessInput(input, stopToken);

                if (eof) {
                    CloseOutput(State::Finished);
                    return;
                }
            }
        }
    } catch (...) {
        m_workerException = std::current_exception();
        CloseOutput(State::Failed);
    }
}
```

Универсальная обработка пакетов и фреймов

```
void ProcessInput(const Input& input, std::stop_token stopToken) {
    while (m_processor.Send(input) == SendResult::NeedReceive) { // Дренируем, пока не получится отправить входные данные
        DrainProcessor(stopToken);
    }
    DrainProcessor(stopToken);
}

void DrainProcessor(std::stop_token stopToken) {
    Output output;

    while (m_processor.Receive(output) == ReceiveResult::Produced) { // Забираем все готовые результаты
        m_outputQueue.PushOrWait(std::move(output), stopToken);
        m_pipelineNotifier.Notify(); // Уведомляем оркестратор об изменении состояния

        output = {};
    }
}

void CloseOutput(State state) {
    m_state.store(state, std::memory_order_release);
    m_outputQueue.Close();
    m_pipelineNotifier.Notify();
}
```

```
export using AsyncEncoder = AsyncStage<Encoder, ffmpeg::Frame, ffmpeg::Packet>;
export using AsyncDecoder = AsyncStage<Decoder, ffmpeg::Packet, ffmpeg::Frame>;
```

Уведомление об изменении состояния стадии

```
export struct PipelineNotifier {
    std::mutex mutex;
    std::condition_variable cv;
    std::atomic_uint64_t epoch = 0;

    void Notify() {
        epoch.fetch_add(1, std::memory_order_relaxed);
        cv.notify_one();
    }

    void Wait(std::uint64_t oldEpoch) {
        if (epoch.load(std::memory_order_relaxed) != oldEpoch) return;
        std::unique_lock lock{ mutex };
        cv.wait(lock, [&] {
            return epoch.load(std::memory_order_relaxed) != oldEpoch;
        });
    }
};
```

Асинхронный транскодер

```
export class AsyncTranscoder {
    struct Branch {
        AsyncDecoder &decoder;
        AsyncEncoder &encoder;
        int streamIndex = -1;
        Muxer::TrackId track;
        bool decoderInputClosed = false;
        bool decoderEof = false;
        bool encoderInputClosed = false;
        bool encoderEof = false;
        std::optional<Packet> pendingPacket;
        std::optional<Frame> pendingFrame;
    };

    Branch m_video;
    Branch m_audio;
    Muxer &m_muxer;
    Demuxer &m_demuxer;
    mm_pipeline::PipelineNotifier &m_pipelineNotifier;
    bool m_demuxEof = false;
public:
    AsyncTranscoder(Muxer &muxer, Demuxer &demuxer, const BranchConfig &video,
                    const BranchConfig &audio, PipelineNotifier &pipelineNotifier);
};
```

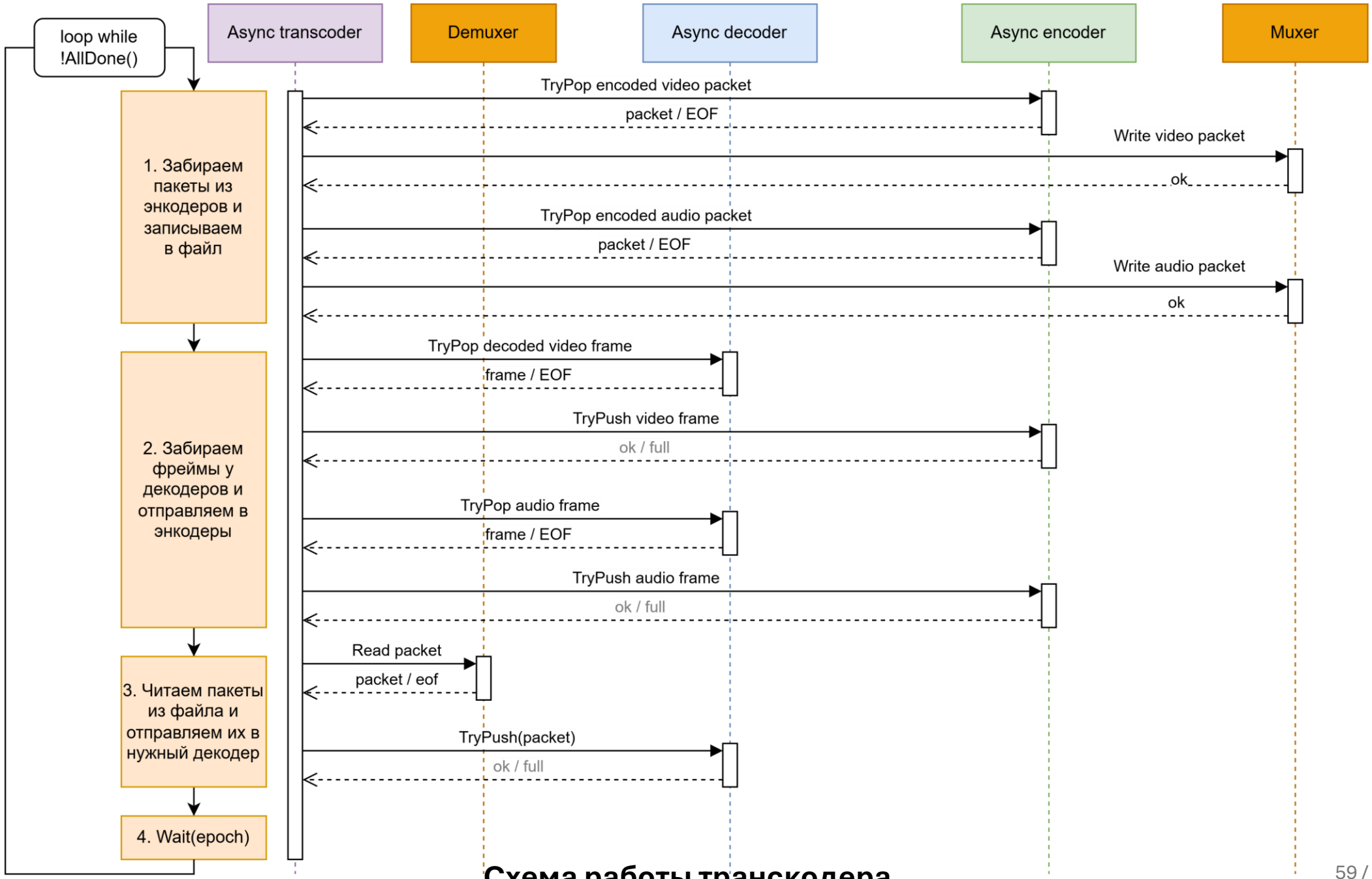


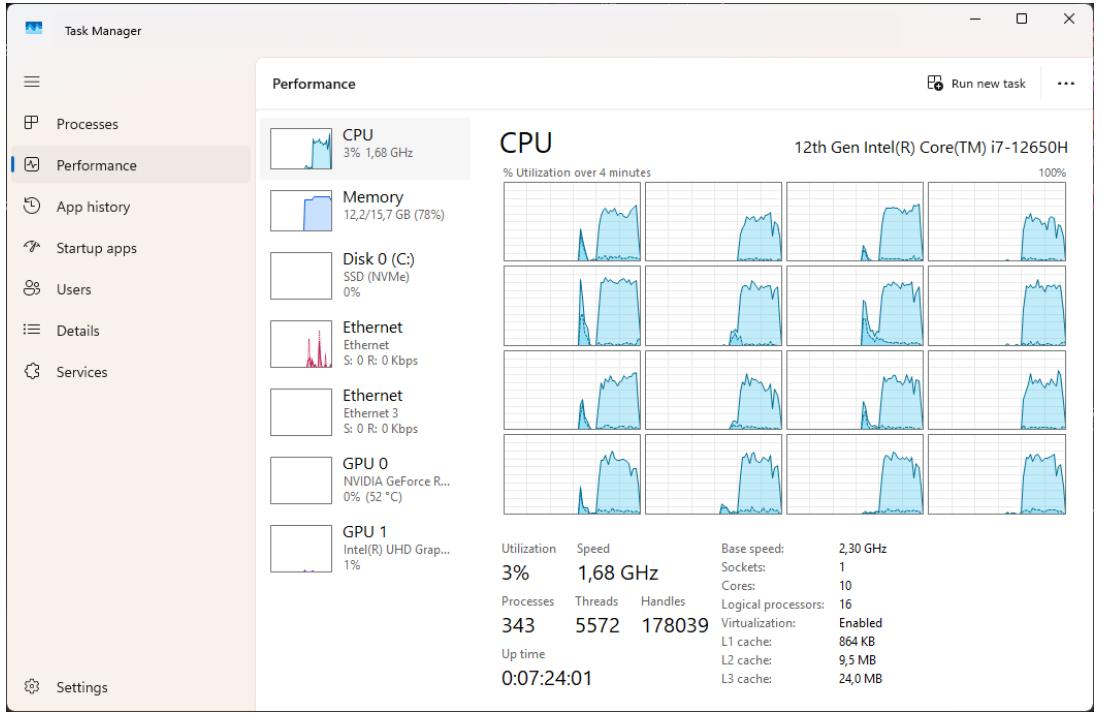
Схема работы транскодера

Рабочий цикл транскодера

```
export class AsyncTranscoder {
    void Run() {
        ... // Initialization and setup
        while (!AllDone()) {
            auto epoch = m_pipelineNotifier.epoch.load(std::memory_order_relaxed);

            bool progress = false;
            progress |= DrainEncoder(m_video);
            progress |= DrainEncoder(m_audio);
            progress |= DrainDecoder(m_video, videoFrameProcessor);
            progress |= DrainDecoder(m_audio, audioFrameProcessor);
            progress |= FeedDemux();

            if (!progress) m_pipelineNotifier.Wait(epoch);
        }
        ... // Shutdown
    }

    bool AllDone() const noexcept { return m_video.encoderEof && m_audio.encoderEof; }
};
```

Оценка производительности

- Скринкаст (14:55):
  - Последовательный конвейер: 108.14 с
  - Параллельный конвейер: 67.67 с (1.6x)
  - FFmpeg: 64.38 (1.68x)
- Динамичный видео-файл (4:33):
  - Последовательный конвейер: 95.40 с
  - Параллельный конвейер: 85.91 с (1.11x)
  - FFmpeg: 68.46 с (1.39x)



Загрузка процессора при использовании параллельного конвейера

Итоги

- FFmpeg + C++23 = ❤️‍🩹
- RAII сильно облегчают жизнь
- Удалось ускорить конвейер за счёт распараллеливания независимых этапов
- FFmpeg пока перегнать не удалось 😞
  - Но это тема для следующего доклада
- Перспективы на будущее:
  - Рассмотреть применение coroutines, generators, ranges

Спасибо за внимание

Telegram: https://t.me/vivid_coding

YouTube: https://youtube.com/@vividbw

Демо код транскодера: https://github.com/alexey-malov/ffmpeg-transcoder-demo