

Universidade Federal de Juiz de Fora
Faculdade de Engenharia e Instituto de Ciências Exatas
Bacharelado em Engenharia Computacional

Pedro Leopoldo Dal-Col Vianna

**Modelo arquitetural baseado no NBMP para workflows de mídia com
suporte a funções de IA**

Juiz de Fora
2026

Pedro Leopoldo Dal-Col Vianna

**Modelo arquitetural baseado no NBMP para workflows de mídia com
suporte a funções de IA**

Trabalho de Conclusão de Curso apresentado ao corpo docente do curso de graduação em Engenharia Computacional da Faculdade de Engenharia e Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do grau de Bacharel em Engenharia Computacional.

Orientador: Prof. D.Sc. Marcelo Moreno

Juiz de Fora

2026

Pedro Leopoldo Dal-Col Vianna,

Modelo arquitetural baseado no NBMP para workflows de mídia com suporte a funções de IA/ Pedro Leopoldo Dal-Col Vianna. - - 2026.

XIII, 57 p.: il.; 29, 7cm.

Orientador: Marcelo Moreno

Trabalho de Conclusão de Curso (bacharelado) – Universidade Federal de Juiz de Fora, ICE/Engenharia. Bacharelado em Engenharia Computacional, 2026.

Referências Bibliográficas: p. 55 – 57.

1. NBMP. 2. Processamento de mídia em rede. 3. Arquitetura de software. 4. Workflow de mídia. 5. Funções de Inteligência Artificial. I. , Marcelo Moreno. II. Universidade Federal de Juiz de Fora, Bacharelado em Engenharia Computacional.

Pedro Leopoldo Dal-Col Vianna

**Modelo arquitetural baseado no NBMP para workflows de mídia com
suporte a funções de IA**

Trabalho de Conclusão de Curso apresentado
ao corpo docente do curso de graduação em
Engenharia Computacional da Faculdade de
Engenharia e Instituto de Ciências Exatas
da Universidade Federal de Juiz de Fora
como requisito parcial à obtenção do grau de
Bacharel em Engenharia Computacional.

Aprovado em 15 de julho de 2026.

BANCA EXAMINADORA

Prof. D.Sc. Marcelo Moreno - Orientador
Universidade

Prof. D.Sc. Jairo
Universidade

Prof. D.Sc. Eduardo Pagani
Universidade

*Aos amigos e familiares, pelo
suporte durante toda a graduação.*

AGRADECIMENTOS

A Deus, em primeiro lugar, que sempre me iluminou, intercedeu e me trouxe aprendizados de humildade, compaixão e amor.

Ao meu professor orientador, Marcelo Moreno, que durante a construção deste trabalho me deu todo o apoio intelectual e incentivo.

Aos meus pais, Armando e Cândida, pelo suporte financeiro e emocional durante toda essa trajetória, sem eles nada disso seria possível.

À minha irmã, Carolina, pelo carinho, amizade e inspiração na vida acadêmica.

Aos meus padrinhos, Claudio e Keila, pelos incentivos e apoio durante toda minha formação educacional.

À minha namorada, Laura Bento, por todo o suporte, carinho, paciência e apoio durante todos os momentos dessa jornada.

Aos meus amigos da Faculdade, que me acompanharam nos momentos de dificuldade e de alegria.

*“Minha verdadeira vantagem
nunca foi meu intelecto incomum.
Tem sido a minha capacidade
de encontrar as pessoas certas e
ficar perto delas.”*

(Brandon Sanderson)

RESUMO

O padrão MPEG-NBMP (*Network-Based Media Processing*, ISO/IEC 23090-8) define um modelo para descrever e executar o processamento de mídia distribuído em rede, mas o faz como um documento normativo extenso e essencialmente textual, sem uma arquitetura de referência diretamente implementável. Soma-se a isso a dificuldade de reusar funções de processamento — em especial as baseadas em Inteligência Artificial (IA) —, frequentemente desenvolvidas acopladas a um único fluxo. Este trabalho propõe um modelo arquitetural derivado do NBMP para a execução de *workflows* de mídia com suporte ao reúso de funções de IA. O padrão foi traduzido em um modelo de classes na notação UML, posteriormente materializado em uma prova de conceito: um orquestrador genérico, implementado em Python com Docker, que lê os descritores do NBMP e executa o *pipeline* descrito, sem lógica dedicada a cada caso. A arquitetura foi avaliada por meio de dois casos de uso de domínios distintos — reconhecimento facial em vídeo e leitura de números em *display* de sete segmentos —, ambos com funções centrais baseadas em IA. O mesmo orquestrador executou os dois casos sem nenhum desvio condicional por caso de uso, com toda a diferença entre eles vivendo nos descritores, e integrou as funções de IA como funções NBMP comuns. Conclui-se que o modelo derivado do NBMP é executável e genérico, e que a separação entre orquestração e descrição permite recompor funções de IA em diferentes *pipelines* sem alterar o código.

Palavras-chave: NBMP. Processamento de mídia em rede. Arquitetura de software. Workflow de mídia. Funções de Inteligência Artificial.

ABSTRACT

The MPEG-NBMP standard (*Network-Based Media Processing*, ISO/IEC 23090-8) defines a model to describe and execute media processing distributed over a network, but it does so as a lengthy, essentially textual normative document, without a reference architecture that is directly implementable. In addition, reusing processing functions — especially those based on Artificial Intelligence (AI) — is difficult, since they are often developed coupled to a single flow. This work proposes an architectural model derived from NBMP for executing media *workflows* with support for the reuse of AI functions. The standard was translated into a class model in UML notation, later materialized as a proof of concept: a generic orchestrator, implemented in Python with Docker, that reads the NBMP descriptors and executes the described *pipeline*, with no logic dedicated to each case. The architecture was assessed through two use cases from distinct domains — facial recognition in video and reading numbers on a seven-segment *display* —, both with core functions based on AI. The same orchestrator executed both cases without any conditional branch per use case, with all the difference between them living in the descriptors, and integrated the AI functions as ordinary NBMP functions. It is concluded that the model derived from NBMP is executable and generic, and that the separation between orchestration and description allows AI functions to be recomposed in different *pipelines* without changing the code.

Keywords: NBMP. Network-based media processing. Software architecture. Media workflow. Artificial Intelligence functions.

LISTA DE ILUSTRAÇÕES

2.1	Arquitetura de referência do NBMP, com seus atores e os fluxos de controle e de dados. Fonte: Neugebauer (Neugebauer, 2025).	19
3.1	Fases do trabalho: estudo (linear), modelagem iterativa (ciclo pergunta-âncora) e implementação.	24
4.1	Visão geral da arquitetura proposta, organizada em seis grupos de classes. . .	28
4.2	Recorte do modelo: o grupo Função (FDD).	30
4.3	Recorte do modelo: o grupo Conectores do Grafo.	33
4.4	Recorte do modelo: o grupo Grafo de Tarefas (WDD).	34
4.5	Recorte do modelo: o grupo Configuração e Requisitos, compartilhado por função e tarefa.	36
4.6	Recorte do modelo: o grupo Processamento e Infraestrutura (MPE).	37
4.7	Recorte do modelo: o grupo Workflow e Orquestração.	38
5.1	Organização do repositório da implementação em três frentes principais. . . .	40
5.2	Trecho esquemático de um FDD: a descrição de uma função isolada.	42
5.3	Trecho esquemático de um WDD: a montagem do <i>pipeline</i> no Modo A.	42
5.4	Máquina de estados que governa a execução de cada Tarefa: de pendente a executando e, ao término, aos estados terminais concluída ou falha . . .	45
6.1	<i>Pipeline</i> do Caso 1: reconhecimento facial. A fonte de mídia distribui o vídeo para as três funções; cada seta indica o artefato trafegado pelo volume compartilhado.	47
6.2	<i>Frame</i> do vídeo de saída do Caso 1, com três pessoas identificadas e rotuladas.	48
6.3	<i>Pipeline</i> do Caso 2: leitura de números em <i>display</i> de sete segmentos. A tarefa de F4 é distribuída em 4 MPEs concorrentes, e AgregaMidia reúne os resultados parciais em um único resultado.json	49
6.4	<i>Frame</i> do vídeo de entrada do Caso 2: o <i>display</i> de sete segmentos lido por F4.	51
6.5	Trecho do resultado.json do Caso 2.	51

LISTA DE TABELAS

2.1	Principais descritores de um documento NBMP. Fonte: elaborada a partir de ISO/IEC 23090-8 (ISO/IEC, 2023).	20
4.1	Tradução dos descritores do FDD em elementos do modelo, segundo a régua de fidelidade.	31
5.1	Materialização dos seis grupos do modelo (Capítulo 4) no código do orquestrador.	41
5.2	Mapeamento das chaves dos descritores (vocabulário do padrão) nas classes do modelo.	43

LISTA DE ABREVIACÕES E SIGLAS

DAG	<i>Directed Acyclic Graph</i>
FDD	<i>Function Description Document</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IA	Inteligência Artificial
IEC	<i>International Electrotechnical Commission</i>
ISO	<i>International Organization for Standardization</i>
JSON	<i>JavaScript Object Notation</i>
MPE	<i>Media Processing Entity</i>
MPEG	<i>Moving Picture Experts Group</i>
NBMP	<i>Network-Based Media Processing</i>
PoC	<i>Proof of Concept</i>
SDK	<i>Software Development Kit</i>
TDD	<i>Task Description Document</i>
UML	<i>Unified Modeling Language</i>
WDD	<i>Workflow Description Document</i>
WM	<i>Workflow Manager</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	PROBLEMA E MOTIVAÇÃO	14
1.2	OBJETIVOS	16
1.3	JUSTIFICATIVA	16
1.4	VISÃO GERAL DA METODOLOGIA	17
1.5	ORGANIZAÇÃO DO TRABALHO	17
2	REFERENCIAL TEÓRICO E TRABALHOS RELACIONADOS	18
2.1	O PADRÃO MPEG-NBMP: VISÃO GERAL E ATORES	18
2.2	DESCRITORES DO NBMP	20
2.3	MODOS DE OPERAÇÃO E CICLO DE VIDA DO WORKFLOW	21
2.4	O PADRÃO ARQUITETURAL PIPES AND FILTERS	21
2.5	TRABALHOS RELACIONADOS	22
3	METODOLOGIA.....	24
3.1	ESTUDO DO PADRÃO E TRABALHOS RELACIONADOS	24
3.2	MODELAGEM ITERATIVA	25
3.3	IMPLEMENTAÇÃO	26
4	ARQUITETURA PROPOSTA.....	27
4.1	DECISÕES DE MODELAGEM	29
4.2	FUNÇÃO	30
4.3	CONECTORES DO GRAFO	32
4.4	GRAFO DE TAREFAS	34
4.5	CONFIGURAÇÃO E REQUISITOS	35
4.6	PROCESSAMENTO E INFRAESTRUTURA	36
4.7	WORKFLOW E ORQUESTRAÇÃO	37

5	IMPLEMENTAÇÃO	39
5.1	TECNOLOGIAS E ORGANIZAÇÃO	39
5.2	DO MODELO AO CÓDIGO	40
5.3	OS DESCRITORES EM TEMPO DE EXECUÇÃO	41
5.4	O FLUXO DE EXECUÇÃO	43
5.5	MÁQUINA DE ESTADOS	44
5.6	FRONTEIRAS DESTA VERSÃO	45
6	CASOS DE USO E RESULTADOS	46
6.1	CASO 1: RECONHECIMENTO FACIAL	46
6.2	CASO 2: NÚMEROS EM <i>DISPLAY</i> DE SETE SEGMENTOS	48
6.3	SÍNTESE DOS RESULTADOS	51
7	CONCLUSÃO E TRABALHOS FUTUROS.....	52
7.1	CONTRIBUIÇÕES	52
7.2	LIMITAÇÕES DO MODELO	53
7.3	TRABALHOS FUTUROS	54

1 INTRODUÇÃO

A produção e o consumo de conteúdo multimídia cresceram a ponto de tornar o processamento de mídia uma operação intensiva e onipresente na infraestrutura de rede. Em vez de ocorrer apenas no dispositivo final, tarefas como transcodificação, análise e enriquecimento de áudio e vídeo migraram para ambientes de nuvem e de borda, onde podem ser executadas de forma escalável e próxima da origem ou do destino dos dados (Martín-Pérez *et al.*, 2019).

Esses fluxos tornaram-se também mais heterogêneos. Um mesmo fluxo de processamento, ou *pipeline*¹, pode combinar operações clássicas, como decodificação, redimensionamento e filtragem, com funções baseadas em Inteligência Artificial (IA), como detecção e classificação de objetos, super-resolução e análise de conteúdo (Guedes, 2025). Essa diversidade de funções, frequentemente desenvolvidas de forma independente e acoplada, dificulta o reuso e a interoperabilidade entre diferentes fluxos e plataformas.

Para padronizar a forma como esses fluxos de processamento de mídia são descritos, distribuídos e executados em rede, o grupo MPEG (*Moving Picture Experts Group*) especificou o NBMP (*Network-Based Media Processing*), publicado como a norma ISO/IEC 23090-8 (ISO/IEC, 2023). O padrão define um modelo de processamento de mídia em rede composto por atores bem delimitados e por um conjunto de descritores que especificam, em formato JSON, como um fluxo de processamento deve ser construído e executado. Os atores são a fonte de mídia (*Source*), o gerenciador de fluxo (*Workflow Manager*), as tarefas (*Task*), a entidade de processamento de mídia (*Media Processing Entity*, MPE) e o consumidor (*Sink*).

1.1 PROBLEMA E MOTIVAÇÃO

Funções de processamento de mídia costumam ser desenvolvidas acopladas a um único fluxo, o que dificulta seu reúso, limitação que pesa de forma especial sobre as funções baseadas em IA. São funções de grande potencial de uso (um mesmo

¹Em sentido estrito, *pipeline* designa um encadeamento linear de estágios, enquanto *workflow* denota uma estrutura mais geral, como um grafo direcionado acíclico (DAG). Neste trabalho os dois termos são usados como sinônimos.

reconhecedor, classificador ou filtro de super-resolução serviria a inúmeros fluxos), mas que, desenvolvidas presas a um *pipeline* específico, raramente são reaproveitadas fora dele. O que se desejaria é o oposto: construir a função uma única vez e recompô-la em diferentes *pipelines* apenas alterando a descrição do *workflow*, sem reescrevê-la. Padronizar a forma de descrever e executar esses fluxos, o que o NBMP se propõe a fazer, é justamente o caminho para esse desacoplamento entre a função e o fluxo que a emprega.

Embora completo e rigoroso, o NBMP é um documento normativo, extenso e essencialmente textual. Ele descreve seus descritores e atores em prosa e em tabelas, mas não oferece um modelo arquitetural (um conjunto de decisões de projeto e uma estrutura coesa) que organize esses conceitos de forma compreensível e diretamente implementável. Há, portanto, uma distância considerável entre a leitura da norma e uma realização concreta de um sistema que a siga.

Reduzir essa distância é o que motiva este trabalho. Sem um modelo arquitetural de referência, quem deseja adotar o NBMP precisa reinterpretar do zero um documento normativo extenso, o que eleva a barreira de entrada ao padrão e favorece implementações divergentes. Soma-se a isso o ponto levantado no início da seção: a dificuldade de reusar funções de IA, hoje presas a fluxos específicos, que este trabalho busca enfrentar oferecendo um *workflow* baseado no NBMP no qual essas funções possam ser integradas e recompostas entre diferentes *pipelines*. Mais do que reorganizar o texto da norma, a motivação central é propor decisões arquiteturais concretas para os pontos que o padrão deixa em aberto: escolhas de modelagem fundamentadas na aplicabilidade e no uso técnico, moldadas pelos casos de uso implementados. É essa concretização prática que distingue o modelo proposto de uma simples transcrição do NBMP.

Diante desse contexto, o trabalho estrutura-se em torno da seguinte pergunta:

É possível modelar e implementar uma arquitetura para o processamento de workflows de mídia em rede baseada no padrão MPEG-NBMP com suporte a funções de IA?

1.2 OBJETIVOS

O objetivo geral deste trabalho é construir um modelo arquitetural para a execução de *pipelines* genéricos de processamento de mídia, com suporte ao reúso de funções de IA, fundamentado no padrão MPEG-NBMP, e avaliar essa arquitetura por meio de casos de uso implementados.

Como objetivos específicos, têm-se:

- modelar as entidades e as relações do NBMP em uma arquitetura de software com auxílio de um diagrama de classes na notação *Unified Modeling Language* (UML);
- implementar um *engine* genérico capaz de executar *workflows* a partir de descritores;
- avaliar a arquitetura executando dois casos de uso de domínios distintos, ambos com funções centrais baseadas em IA;
- demonstrar a generalidade da arquitetura: o mesmo *engine* executa *workflows* diferentes trocando apenas os descritores, sem alterar o código;
- demonstrar que funções baseadas em IA podem ser incorporadas e reutilizadas como funções NBMP comuns.

1.3 JUSTIFICATIVA

A principal contribuição deste trabalho está em traduzir um padrão denso e textual em um modelo arquitetural reutilizável e em uma prova de conceito (*Proof of Concept*, PoC) reproduzível. Esse esforço funciona como ponte entre a norma e a prática: reduz a barreira para compreender o NBMP e oferece uma referência concreta para quem deseja implementá-lo.

Além disso, uma arquitetura padronizada e executável reduz o acoplamento entre as partes de um sistema de processamento de mídia e favorece a interoperabilidade e o reúso de funções entre diferentes *workflows*. São as funções de IA, de amplo potencial de uso mas frequentemente presas a um único fluxo, as que mais se beneficiam de poder ser recompostas em vários *pipelines*. O modelo as acomoda como funções comuns, sem

tratamento específico, e essa mesma uniformidade pavimenta o caminho para formas mais ricas de integrá-las, como a seleção automática de funções, retomada entre os trabalhos futuros.

1.4 VISÃO GERAL DA METODOLOGIA

Metodologicamente, o trabalho organizou-se em três fases, detalhadas no Capítulo 3. A primeira foi o estudo do padrão MPEG-NBMP e dos trabalhos relacionados. Na segunda, o modelo arquitetural foi construído de forma iterativa, refinado a cada ciclo por exercícios de aplicabilidade nos casos de uso e por revisões com o orientador. Na terceira, com o modelo já consolidado, ele foi materializado em código de um *engine* como prova de conceito, cuja instanciação, aplicada à execução de dois casos de uso distintos, apoiou a avaliação da arquitetura.

1.5 ORGANIZAÇÃO DO TRABALHO

O restante deste trabalho está organizado como segue. O Capítulo 2 apresenta o referencial teórico sobre o NBMP e os trabalhos relacionados. O Capítulo 3 descreve a metodologia adotada. O Capítulo 4 detalha a arquitetura proposta e sua modelagem, núcleo da contribuição deste trabalho. O Capítulo 5 trata da implementação do *engine* genérico. O Capítulo 6 apresenta os casos de uso e os resultados obtidos. Por fim, o Capítulo 7 reúne as conclusões e os trabalhos futuros.

2 REFERENCIAL TEÓRICO E TRABALHOS RELACIONADOS

Este capítulo reúne os fundamentos necessários à compreensão do restante do trabalho. Apresenta o padrão MPEG-NBMP (seus atores, descritores e modos de operação), o padrão arquitetural *Pipes and Filters*, que inspira parte da modelagem proposta, e os trabalhos relacionados que implementam ou avaliam o NBMP, situando a contribuição desta monografia.

2.1 O PADRÃO MPEG-NBMP: VISÃO GERAL E ATORES

O NBMP é um *framework* padronizado pelo grupo MPEG para descrever e executar o processamento de mídia distribuído em uma rede (ISO/IEC, 2023). Seu objetivo é permitir que um cliente especifique, de forma independente de fornecedor, um fluxo de processamento (por exemplo, transcodificar um vídeo, extrair legendas ou analisar seu conteúdo) e que esse fluxo seja instanciado e executado sobre recursos computacionais distribuídos, sem que o cliente precise conhecer os detalhes de implementação de cada etapa.

A arquitetura de referência do NBMP organiza-se em torno de cinco atores principais, ilustrados na Figura 2.1. A fonte de mídia (*Media Source*) fornece o conteúdo de entrada; o gerenciador de fluxo (*Workflow Manager*, WM) recebe do cliente (*NBMP Client*) uma descrição do que se deseja executar, interpreta-a, seleciona e instancia as funções necessárias e orquestra sua execução; as tarefas (*Tasks*) realizam o processamento dentro de entidades de processamento de mídia (*Media Processing Entities*, MPEs); e o destino (*Media Sink*) consome a mídia produzida. O fluxo de controle conecta cliente, WM e tarefas, enquanto o fluxo de dados percorre da fonte ao destino, passando pelas MPEs.

O restante desta seção detalha cada elemento da arquitetura, no nível conceitual em que o padrão os define.

A ***Media Source*** é a fronteira de entrada do sistema: fornece a mídia a ser processada e, junto ao cliente, sinaliza ao gerenciador o que precisa ser feito sobre ela. O padrão a trata de forma abstrata: importa o conteúdo que ela disponibiliza, não os detalhes de como o produz.

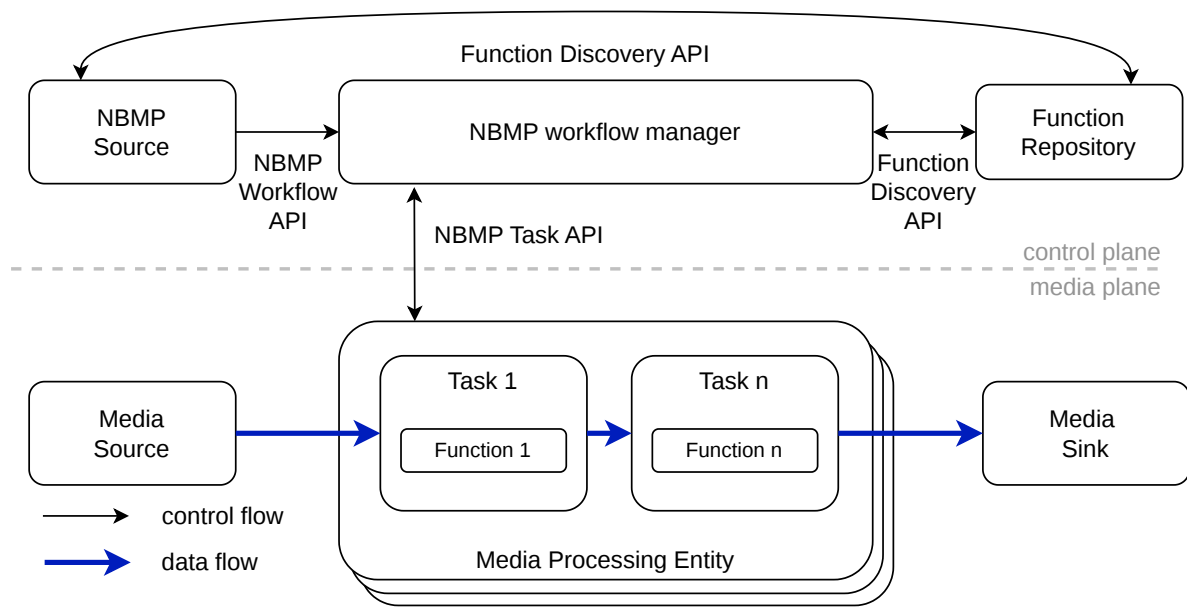


Figura 2.1: Arquitetura de referência do NBMP, com seus atores e os fluxos de controle e de dados. Fonte: Neugebauer (Neugebauer, 2025).

O **NBMP Client** é quem inicia o processo: monta a descrição do *workflow* desejado e a submete ao gerenciador. Já o **Workflow Manager** (WM) é o orquestrador central do NBMP. A partir dessa descrição, ele identifica e instancia as funções necessárias, atribui cada uma a um ambiente de execução, constrói o grafo que liga as etapas e conduz toda a execução, iniciando, monitorando e finalizando o *workflow*. O WM é definido pelo padrão por seu comportamento, e não por um documento de descrição próprio.

Uma **Function** é uma unidade de processamento reutilizável. O NBMP a define como a soma de uma implementação (tipicamente uma imagem de *container*) e de uma descrição (seção 3.1.13); essa descrição declara as entradas, saídas, requisitos e parâmetros da função (seção 6.1.2) (ISO/IEC, 2023). A função é um modelo estático: existe independentemente de qualquer execução e pode ser empregada em diferentes *workflows*.

Uma **Task** é a instância concreta de uma função em tempo de execução. Enquanto a função é o molde, a tarefa é a função em ação: tem estado próprio e ligações de entrada e saída atribuídas pelo WM. Cada tarefa tem sua própria descrição, espelho de tempo de execução da descrição da função. Essa distinção entre função e tarefa estrutura a modelagem proposta no Capítulo 4.

A **Media Processing Entity** (MPE) é o ambiente que hospeda e executa as tarefas. Ela expõe ao WM suas *capabilities* (os recursos e as funções que é capaz de oferecer), e é com base nelas que o gerenciador decide qual tarefa atribuir a qual MPE.

Por fim, a **Media Sink** é a fronteira de saída: recebe e consome a mídia produzida pelo *workflow*, durante ou após a execução. Assim como a fonte, é tratada de forma abstrata pelo padrão.

2.2 DESCRITORES DO NBMP

O NBMP é dirigido por documentos descritivos em formato JSON, que especificam de maneira declarativa o que deve ser executado. Três deles são centrais para este trabalho: o WDD (*Workflow Description Document*), que descreve o *workflow* como um todo; o FDD (*Function Description Document*), que descreve uma função; e o TDD (*Task Description Document*), que descreve uma tarefa em execução (ISO/IEC, 2023).

Cada documento é composto por um conjunto de *descritores* (blocos que agrupam parâmetros afins), resumidos na Tabela 2.1. Os principais são o *General* (identificação), o *Input* e o *Output* (portas e metadados da mídia de entrada e de saída), o *Processing* (a implementação e o grafo de conexões), o *Requirements* (recursos exigidos) e o *Configuration* (parâmetros ajustáveis).

Tabela 2.1: Principais descritores de um documento NBMP. Fonte: elaborada a partir de ISO/IEC 23090-8 (ISO/IEC, 2023).

Descritor	Papel
<i>General</i>	Identificação (id, nome, descrição, palavras-chave).
<i>Input / Output</i>	Portas e metadados da mídia de entrada e de saída.
<i>Processing</i>	Implementação da função e grafo de conexões (<i>connection-map</i>).
<i>Requirements</i>	Recursos exigidos (hardware, latência, vazão).
<i>Configuration</i>	Parâmetros ajustáveis da função.

Os três documentos se relacionam diretamente. O FDD é o molde de uma função, publicado e reutilizável; o WDD, montado pelo cliente, referencia as funções desejadas e

descreve como elas se conectam por meio de um *connection-map*; e o TDD é o espelho de tempo de execução do FDD: o WM o cria ao instanciar cada função como uma tarefa concreta (seção 5.3.2) (ISO/IEC, 2023).

2.3 MODOS DE OPERAÇÃO E CICLO DE VIDA DO WORKFLOW

O NBMP admite duas formas de o cliente construir o grafo de processamento (ISO/IEC, 2023). Na primeira, o cliente especifica explicitamente quais funções compõem o *workflow* e como elas se conectam, fornecendo o *connection-map* diretamente no WDD. Na segunda, o cliente descreve apenas o resultado desejado por meio de palavras-chave, e cabe ao WM descobrir e selecionar, em um repositório, as funções adequadas. Por conveniência de referência, a primeira é denominada Modo A e a segunda, Modo B. Este trabalho adota o Modo A: ele torna o grafo explícito e dispensa um mecanismo de descoberta, mantendo o escopo concentrado na execução do *workflow*, e não na seleção automática de funções.

Uma vez recebido o WDD, o ciclo de vida do *workflow* é conduzido pelo WM em uma sequência bem definida: ele recebe a descrição, instancia as tarefas a partir das funções referenciadas, atribui cada tarefa a uma MPE capaz de executá-la, constrói o grafo de tarefas que representa o fluxo de dados, um grafo acíclico dirigido (*Directed Acyclic Graph*, DAG), e dispara a execução, monitorando-a até a conclusão (ISO/IEC, 2023). Essa sequência de responsabilidades do WM é a base da camada de orquestração detalhada no Capítulo 4.

2.4 O PADRÃO ARQUITETURAL PIPES AND FILTERS

O padrão arquitetural *Pipes and Filters* organiza um sistema que processa um fluxo de dados como uma sequência de componentes de processamento independentes (os *filters*) conectados por canais (os *pipes*) (Buschmann *et al.*, 2008). Cada filtro recebe dados por suas entradas, processa-os e os entrega às saídas, sem conhecer a identidade dos filtros vizinhos: a comunicação ocorre exclusivamente pelos canais. Esse isolamento favorece o reuso dos filtros, a recombinação do fluxo e a substituição de componentes sem impacto sobre os demais.

A semelhança com o NBMP é direta: tarefas que processam mídia e se comunicam por portas e conexões espelham filtros ligados por canais. Este trabalho explora essa correspondência ao modelar explicitamente as classes **Porta** e **Conexao** (Capítulo 4) como elementos isolados (o que, no padrão, é descrito apenas como parâmetros do JSON), tornando mais nítida no modelo arquitetural a característica de *Pipes and Filters*.

2.5 TRABALHOS RELACIONADOS

O NBMP, embora recente, já motivou implementações e estudos que ajudam a situar a contribuição deste trabalho. Três deles são particularmente relevantes, cada um com um foco distinto.

Neugebauer (Neugebauer, 2025) apresenta o *Nagare Media Engine*, uma implementação *open-source* de um gerenciador de *workflows* compatível com o NBMP. O trabalho concentra-se na autoadaptação dos *workflows*: o *engine* é projetado para ajustar dinamicamente a execução em resposta a mudanças nas condições de operação, aproximando o padrão de cenários reais em que a carga e a disponibilidade de recursos variam ao longo do tempo. Por ser aberto e voltado à operação, constitui uma realização madura do NBMP, com ênfase no comportamento em tempo de execução.

Xu et al. (Xu et al., 2022) revisam o progresso recente do uso de computação em nuvem e borda para produção de mídia e apresentam uma implementação baseada no NBMP. O foco do estudo está no desempenho do processamento distribuído: os autores avaliam como diferentes estratégias de implantação (mais centralizadas na nuvem ou mais próximas da borda) afetam o resultado, discutindo o papel do NBMP como camada de padronização sobre uma infraestrutura heterogênea. O trabalho reforça a viabilidade do padrão em ambientes de produção reais.

Ramos-Chavez et al. (Ramos-Chavez et al., 2021) propõem um *testbed* NBMP voltado à avaliação, em escala, de *workflows* de processamento de mídia distribuído em tempo real. O ambiente inclui os componentes de referência do padrão e um conjunto de funções configuráveis para geração de carga, monitoramento, coleta de dados e visualização, permitindo testar configurações de forma padronizada e reproduzível. Em

seus experimentos, os autores executaram uma extensa bateria de testes com funções de *streaming* adaptativo sob diferentes cargas, analisando desempenho, confiabilidade e consistência temporal, e chegando a desenvolver um classificador para detectar falhas sob determinadas condições. É um trabalho centrado na validação experimental do NBMP em larga escala.

Os três trabalhos compartilham um foco em implementar ou avaliar *engines* NBMP em cenários de operação. A contribuição desta monografia situa-se em outro eixo: em vez de um *engine* de produção, propõe-se um modelo arquitetural do padrão, mais fácil de compreender e fiel ao NBMP, ainda que com simplificações deliberadas e justificadas (Capítulo 4), avaliado por uma prova de conceito que demonstra sua generalidade: o mesmo *engine* executa *pipelines* distintos apenas trocando os descritores, integrando inclusive funções baseadas em IA como funções NBMP comuns. Reconhece-se, com franqueza, que essa prova de conceito é mínima (de execução local e sequencial), e não um *engine* concorrente em robustez ou escala. Seu valor está em oferecer uma ponte compreensível e implementável entre o texto da norma e a prática, deixando margem para trabalhos futuros como a integração do treinamento e da validação de funções baseadas em *machine learning* e formas mais otimizadas de orquestração e escalabilidade das MPEs.

3 METODOLOGIA

Este trabalho caracteriza-se como pesquisa aplicada de caráter experimental: seu objetivo é projetar, construir e demonstrar a viabilidade de um artefato (um modelo arquitetural e uma prova de conceito executável), e não formular leis universais sobre o domínio. O processo organizou-se em três fases distintas, ilustradas na Figura 3.1: *estudo*, *modelagem iterativa* e *implementação*.

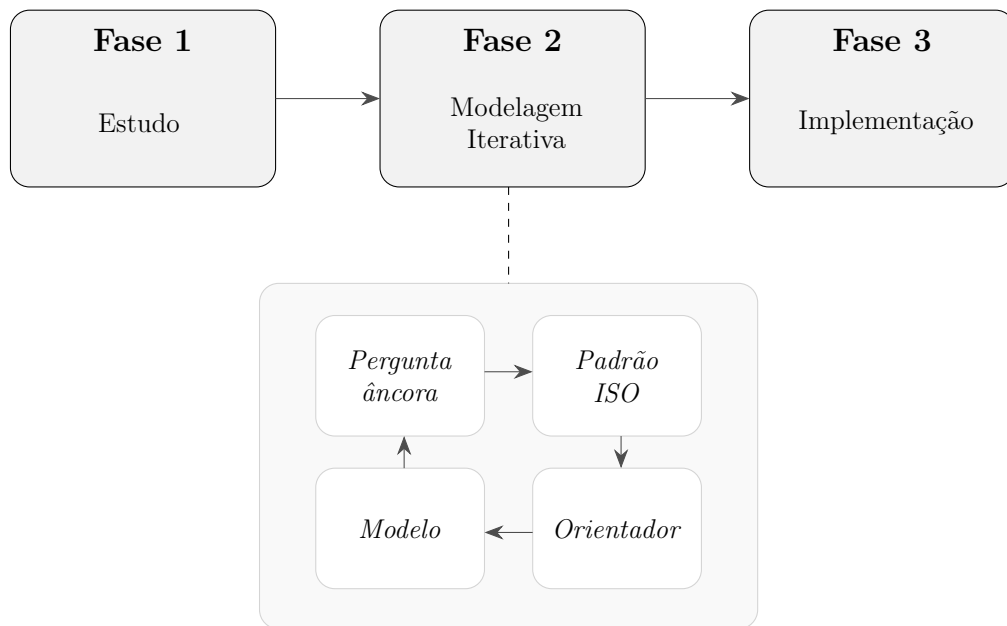


Figura 3.1: Fases do trabalho: estudo (linear), modelagem iterativa (ciclo pergunta-âncora) e implementação.

3.1 ESTUDO DO PADRÃO E TRABALHOS RELACIONADOS

A primeira fase concentrou-se na leitura do padrão MPEG-NBMP (ISO/IEC 23090-8 (ISO/IEC, 2023)) e dos trabalhos relacionados discutidos no Capítulo 2 (Neugebauer, 2025; Xu *et al.*, 2022; Ramos-Chavez *et al.*, 2021). A leitura do padrão foi guiada pela pergunta central do trabalho, como organizar as entidades do NBMP em um modelo de classes coerente e implementável, capaz de sustentar o reúso de funções (em especial as baseadas em IA) entre diferentes *workflows*, e focada nos capítulos de definições, terminologia e arquitetura geral.

O estudo dos trabalhos relacionados situou o trabalho no panorama de realizações

existentes do NBMP. Os três trabalhos investigados concentram-se, cada um, em um aspecto diferente: autoadaptação de *workflows*, desempenho em cenários de nuvem e borda, e avaliação de sistemas em escala. Esse mapeamento situou o recorte deste trabalho: enquanto os estudos investigados enfatizam desempenho, escala ou autoadaptação de sistemas em operação, o foco aqui é a construção de um *modelo arquitetural* compreensível e de uma prova de conceito que demonstre a generalidade da arquitetura, inclusive no reúso de funções de IA como funções comuns, um enfoque complementar ao das realizações estudadas.

3.2 MODELAGEM ITERATIVA

A segunda fase foi a mais longa e central do trabalho: a construção iterativa do modelo arquitetural. Cada iteração partia de uma pergunta-âncora: uma questão específica sobre como um conceito do padrão deveria ser representado no modelo. A resposta era buscada diretamente no texto normativo, discutida com o orientador e, então, incorporada ao diagrama de classes. A resposta, por sua vez, gerava novas perguntas, abrindo o próximo ciclo.

Esse processo tornou as decisões de modelagem resistentes à arbitrariedade: ao ancorar cada escolha em uma pergunta concreta e compará-la com o padrão, manteve-se uma linha clara entre interpretação fundamentada e convenção. As decisões de escopo emergiram naturalmente do ciclo. Uma régua de fidelidade guiou o mapeamento de descritores em classes: descritores com estrutura interna rica foram modelados como classes próprias; campos simples ou de valor único foram achatados como atributos da classe-mãe; já os descritores opcionais ou raramente usados ficaram fora do diagrama. Adicionalmente, o escopo foi delimitado ao Modo A de configuração do *Workflow Description Document* (WDD), no qual o cliente especifica as funções e o grafo de conexões (*connection-map*) diretamente no descritor, excluindo o Modo B e os *function groups*, funções compostas que encapsulam sub-*workflows* como uma única entidade. O resultado estabilizado após os ciclos é o diagrama de classes apresentado no Capítulo 4.

3.3 IMPLEMENTAÇÃO

A terceira fase materializou o modelo em código como prova de conceito, e desenvolveu-se em dois momentos complementares.

No primeiro, ainda antes de escrever código, foram realizados exercícios de abstração: para cada caso de uso, identificou-se quais classes do modelo participariam do *pipeline*, como ficariam os grafos de tarefas e quais parâmetros seriam necessários. Esses exercícios funcionaram como uma verificação do modelo em um contexto mais concreto: uma forma de testar, sem ainda escrever código, se a abstração construída na fase anterior se sustentava na prática. Ao longo deles, lacunas e imprecisões foram identificadas e corrigidas antes de qualquer implementação.

No segundo momento, o *engine* genérico foi implementado em Python 3.11 com o *Docker Software Development Kit* (SDK) *for Python*. O *runtime* de cada função é um container Docker (uma imagem por função), e o processamento de mídia e inteligência artificial utiliza bibliotecas consolidadas: OpenCV, face_recognition, FFmpeg e ssocr. Todas as escolhas seguiram o princípio de reuso de ferramentas prontas, sem treinamento de modelos próprios, princípio que permite ao modelo aproveitar funções de IA já disponíveis de forma genérica, incorporando-as como funções comuns sem qualquer treino. O modelo, já consolidado pelos ciclos iterativos e pelos exercícios de abstração, mostrou-se suficientemente robusto para que a implementação avançasse sem exigir alterações na arquitetura. O mesmo *engine* genérico executou os dois casos de uso (reconhecimento facial em vídeo e leitura de números em *display* de sete segmentos) sem desvio condicional por caso de uso, avaliando a generalidade da arquitetura proposta. Como as funções centrais dos dois casos são baseadas em IA, essa avaliação abrange também o reuso de funções de IA entre *pipelines* distintos.

4 ARQUITETURA PROPOSTA

Este capítulo apresenta a arquitetura proposta para um sistema NBMP, resultado da interpretação do padrão em um modelo de classes. A arquitetura não se limita a transcrever o texto da norma: ela organiza os conceitos do NBMP em um conjunto coeso de classes e relações e registra decisões de projeto (o que merece existir como entidade própria, como os elementos se associam e quais detalhes são essenciais), de modo a tornar o padrão mais compreensível e diretamente implementável.

A Figura 4.1 oferece uma visão de alto nível do modelo, organizado em seis grupos de classes. No topo está o grupo Workflow e Orquestração: o `WorkflowManager` recebe a descrição do *workflow* e coordena toda a execução, gerenciando um `Workflow`. A partir dessa descrição, monta-se o grupo Grafo de Tarefas (correspondente ao WDD), um agregador das classes que se relacionam para formar o *pipeline* a ser executado em tempo de execução: cada `Tarefa` representa uma etapa de processamento, e as fronteiras de entrada e saída são as classes `Source` e `Sink`. Cada tarefa instancia uma `Funcao`, definida no grupo Função (correspondente ao FDD) junto de suas descrições, como a `Imagem` que a implementa. As tarefas executam sobre o grupo Processamento e Infraestrutura, que representa a entidade de processamento, a classe `MPE`, descrita por suas `Capabilities`. A ligação entre os nós do grafo ocorre pelo grupo Conectores do Grafo (`Porta`, `Conexao` e `MidiaMetadata`), que dá forma ao fluxo de dados. Por fim, o grupo Configuração e Requisitos, compartilhado por função e tarefa, declara os parâmetros ajustáveis e os recursos exigidos.

Em toda a apresentação, o modelo segue as convenções da *Unified Modeling Language* (UML): as classes aparecem como retângulos com seus atributos, e as relações como associações ou composições, estas últimas indicando que uma classe é parte indissociável de outra. Os nomes seguem uma convenção uniforme: classes em *PascalCase* e atributos em *camelCase*. Termos consagrados do padrão MPEG-NBMP permanecem em inglês (como `Source`, `Sink` e `Capabilities`); termos próprios do projeto são nomeados em português (como `Funcao`, `Tarefa` e `Porta`).

A próxima seção apresenta as decisões de modelagem que orientaram a arquitetura;

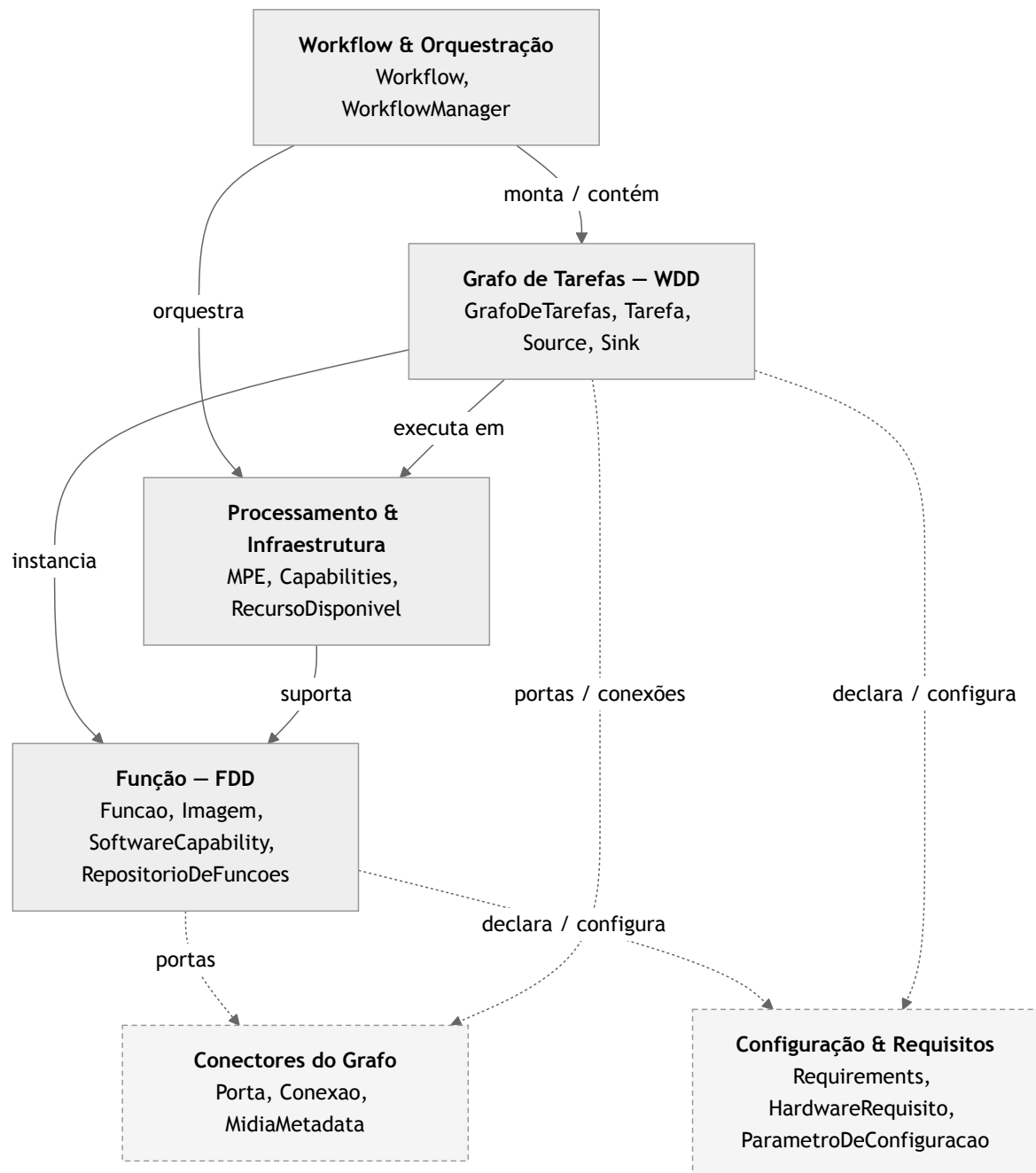


Figura 4.1: Visão geral da arquitetura proposta, organizada em seis grupos de classes.

em seguida, cada grupo é detalhado em uma seção própria, acompanhado de um recorte do diagrama que isola suas classes e relações. O modelo completo é a soma desses recortes. Nesses recortes, as classes com borda tracejada pertencem a outros grupos e aparecem apenas para mostrar as ligações entre eles.

4.1 DECISÕES DE MODELAGEM

Como descrito no Capítulo 2, o NBMP admite dois modos do cliente construir o grafo de processamento. A arquitetura proposta neste trabalho adota o Modo A, em que o cliente especifica as funções e suas conexões diretamente no WDD. Trata-se de uma decisão de escopo: o Modo A é concreto e bem delimitado o bastante para ser implementado por completo, o que viabiliza a entrega de uma prova de conceito efetivamente funcional. O Modo B, ao contrário, depende de um mecanismo de descoberta de funções que o próprio padrão deixa abstrato, como questão de implementação; realizá-lo ampliaria o escopo muito além de uma prova de conceito. Por isso ele é deixado como trabalho futuro. A classe **SoftwareCapability**, cujo propósito de implementação é detalhado adiante, foi concebida justamente como uma base possível para essa futura realização do Modo B. A consequência direta da escolha para a modelagem é que o grafo é dado pelo cliente, e não descoberto: o modelo não precisa de um mecanismo de seleção automática de funções.

Uma segunda decisão, de natureza estrutural, atravessa todo o modelo: manter em classes distintas o que o próprio padrão separa ao definir uma função: a implementação e a descrição. Esse princípio se repete em diferentes pontos. Na definição de funções, a **Funcao** (a descrição reutilizável) é mantida separada da **Imagem** (a sua implementação). Na execução, a **MPE** é mantida mínima, e toda a descrição dos seus recursos vive na classe **Capabilities**. E a relação entre **Funcao** e **Tarefa** segue a mesma lógica de separação: a **Funcao** é a definição estática e reutilizável, enquanto a **Tarefa** é a sua instância em tempo de execução. Tratar esses papéis como classes distintas é o que mantém o modelo coeso, e cada caso é retomado na seção do grupo correspondente.

Por fim, a tradução de cada descritor do padrão em elementos do modelo seguiu a régua de fidelidade descrita no Capítulo 3. Sua aplicação concreta aparece nas seções

seguintes, a começar pela tradução dos descritores do FDD, na próxima seção.

4.2 FUNÇÃO

O grupo Função representa a função do NBMP: a definição reutilizável de uma etapa de processamento. A Figura 4.2 mostra suas classes (*Funcao*, *Imagem*, *SoftwareCapability* e *RepositorioDeFuncoes*) e as ligações com os demais grupos. É aqui que se materializa o FDD, o documento que descreve uma função.

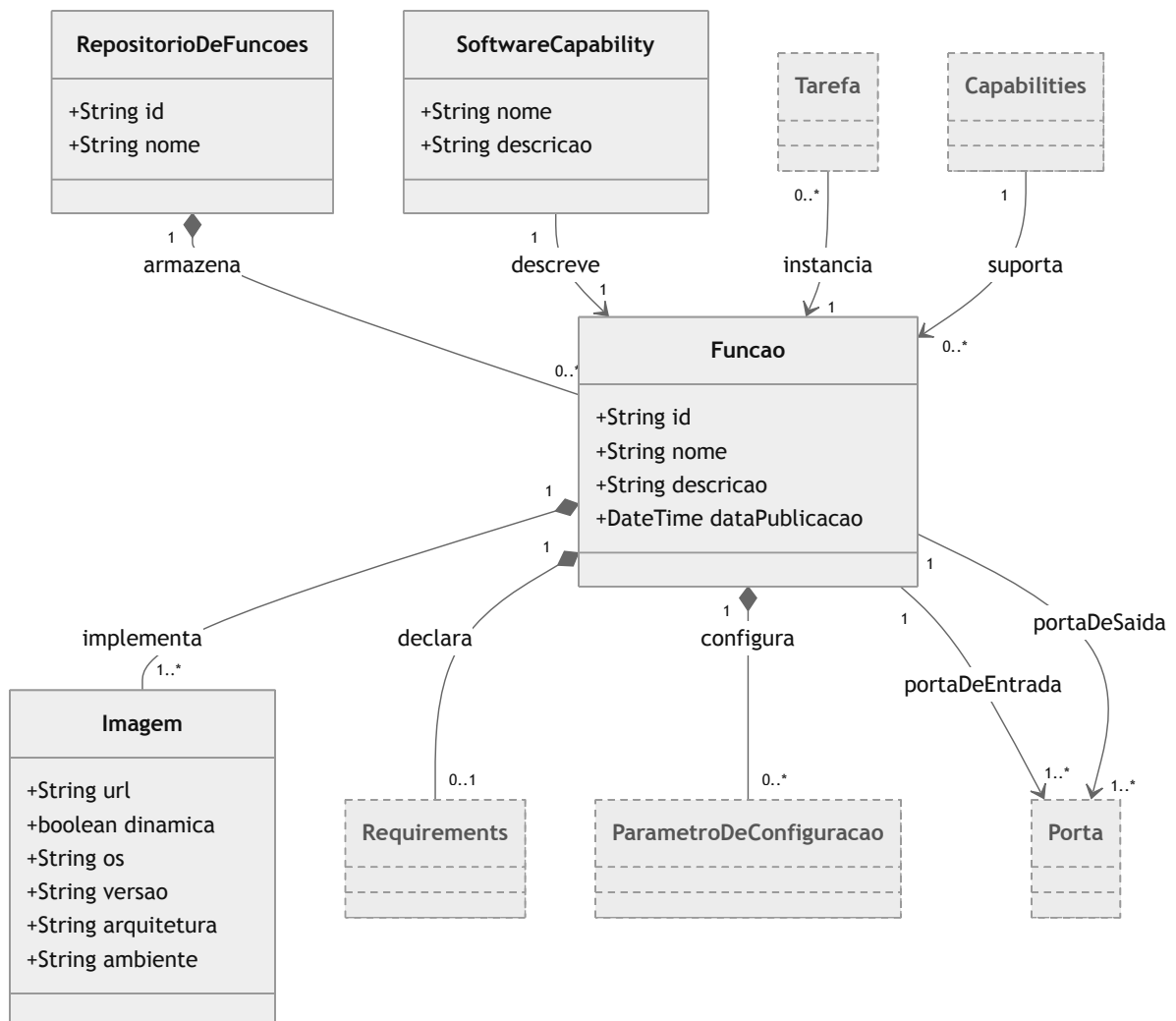


Figura 4.2: Recorte do modelo: o grupo Função (FDD).

A classe *Funcao* é o centro do grupo: representa o *template* de uma função, uma definição estática e reutilizável, independente de qualquer execução. Seus atributos (*id*, *nome*, *descricao*, *dataPublicacao*) a identificam, e, como discutido na seção anterior, ela é o lado da *descrição* no par implementação-descrição do NBMP. Os demais descritores do

FDD foram traduzidos em classes ou atributos segundo a régua de fidelidade; a Tabela 4.1 resume esse mapeamento.

Tabela 4.1: Tradução dos descritores do FDD em elementos do modelo, segundo a régua de fidelidade.

Descritor do FDD		Tratamento	No modelo
General		atributos	atributos da Funcao
Input / Output		classe própria	Porta, MidiaMetadata
Processing		classe própria	Imagem
Requirements		classe própria	Requirements, HardwareRequisito
Configuration		classe própria	ParametroDeConfiguracao
Demais	descritores opcionais	omitido	—

A classe **Imagem** é a implementação da função: uma imagem de *container* que pode ser executada. Seus atributos descrevem essa implementação: a *url* de onde obtê-la, o ambiente esperado (*os*, *versao*, *arquitetura*, *ambiente*) e o atributo *dinamica*. Este último distingue uma imagem estática (pré-construída e já disponível em um registro, bastando obtê-la) de uma imagem dinâmica, resolvida ou gerada em tempo de implantação, como uma imagem construída sob demanda ou cuja localização só é determinada em tempo de execução. A relação **Funcao 1 — 1..* Imagem** indica que uma função tem uma ou mais imagens: ao menos uma para existir como implementação e, eventualmente, várias (por exemplo, uma para cada sistema operacional ou arquitetura de destino). É essa classe que completa a definição NBMP de função como implementação somada a descrição.

A classe **SoftwareCapability** guarda, em linguagem natural, aquilo que uma função é capaz de fazer: a sua semântica. Enquanto os demais descritores dizem *como* a função é executada, ela diz *o que* a função significa. À primeira vista, isso se sobrepõe ao atributo *descricao* da própria **Funcao**, contudo, a razão de existir como classe separada está no

seu papel pretendido, não no seu conteúdo atual. A `descricao` é documentação para o leitor humano, ao passo que a `SoftwareCapability` é concebida para ser interpretada por máquina: é onde, no futuro, a semântica de cada função poderá sustentar a seleção automática de funções (Modo B). Esse desdobramento é tratado na seção de trabalhos futuros no Capítulo 7. A classe é mantida em inglês, por exceção à convenção de nomes, pela proximidade com a classe `Capabilities`.

Por fim, a classe `RepositorioDeFuncoes` é o armazém das funções: mantém as `Funcao` do modelo (composição `1 — 0..*`) e é a origem única dos FDDs. Além das funções específicas de cada caso de uso, guarda um conjunto de funções-base reutilizáveis (como extração de quadros e distribuição de mídia), aproveitáveis por diferentes *workflows*. Vale distinguir esse repositório do mecanismo de descoberta do Modo B: aqui ele é um armazém passivo, consultado pelo nome da função, sem qualquer busca por palavra-chave.

4.3 CONECTORES DO GRAFO

O grupo Conectores do Grafo reúne os elementos que ligam os nós do grafo e dão forma ao fluxo de dados: `Porta`, `Conexao` e `MidiaMetadata`. É por meio deles que a saída de uma etapa de processamento chega à entrada da etapa seguinte. A Figura 4.3 mostra suas classes e as ligações com os nós que expõem portas. No padrão MPEG-NBMP, as portas de entrada e saída e o *connection-map* aparecem apenas como campos em formato JSON dentro dos descritores; na arquitetura proposta, eles são promovidos a classes próprias do modelo. Essa é a aplicação concreta do padrão *Pipes and Filters* discutido no Capítulo 2: as portas são as junções por onde os filtros (as tarefas) se acoplam, e as conexões são os tubos que conduzem a mídia entre eles.

A classe `Porta` representa um ponto de entrada ou de saída de dados de um nó. Seus atributos (`streamId`, `nome`, `protocolo`, `modo`, `cachingServerUrl`) descrevem como a mídia é exposta naquele ponto. A mesma classe serve a todos os nós que participam do fluxo (`Source`, `Sink`, `Tarefa` e `Funcao`); o que distingue uma porta de entrada de uma de saída não é a classe, mas o papel que a relação lhe atribui. Independentemente desse papel, o atributo `modo` descreve uma outra característica da porta: se a mídia é empurrada

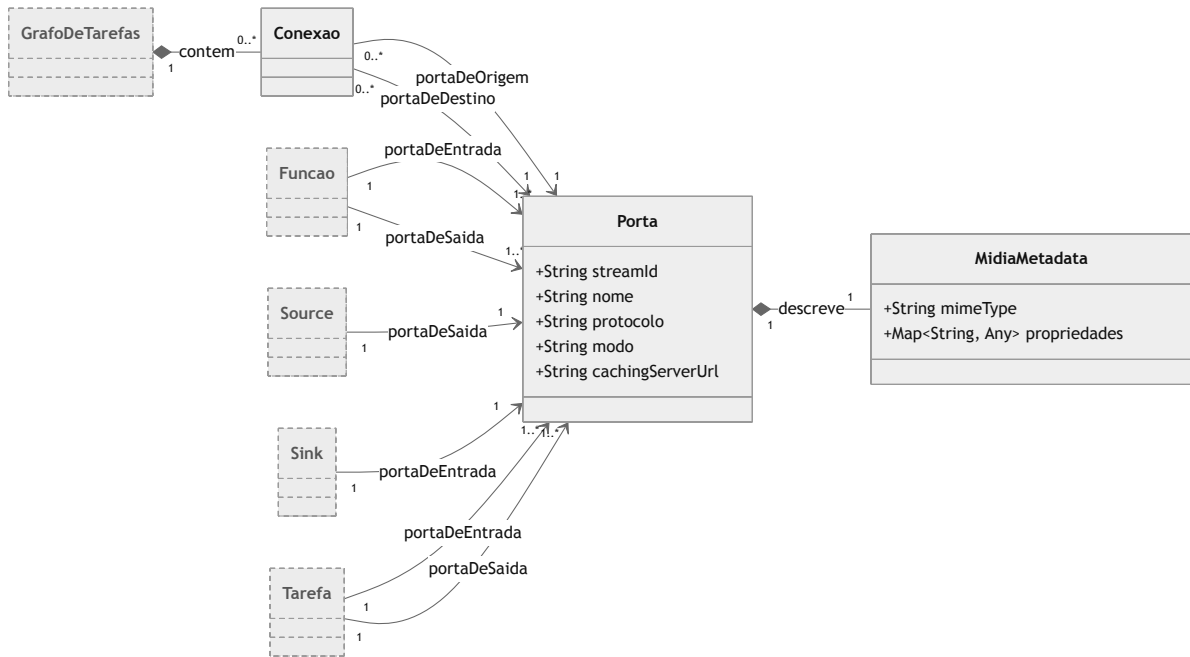


Figura 4.3: Recorte do modelo: o grupo Conectores do Grafo.

pela origem (*push*) ou puxada pelo destino (*pull*). Há ainda uma restrição herdada do padrão: o `streamId` é proibido quando a `Porta` pertence a uma `Funcao` (Tabela 1 do padrão MPEG-NBMP). A razão é que, na função, o fluxo ainda é abstrato (a função é um *template*); o identificador concreto do fluxo só passa a existir quando a função é instanciada como `Tarefa`.

A classe `Conexao` é a ligação dirigida entre duas portas: parte de uma `portaDeOrigem` e chega a uma `portaDeDestino`. É ela que materializa o *connection-map* do WDD e transforma nós isolados em um grafo efetivamente executável. A cardinalidade `0..*` de `Conexao` sobre cada `Porta` permite que uma mesma porta de origem participe de várias conexões: a saída de uma etapa pode, assim, ser distribuída para mais de uma etapa seguinte. É o caso, por exemplo, de um mesmo quadro de vídeo encaminhado simultaneamente a duas tarefas de análise distintas.

Por fim, a classe `MidiaMetadata` descreve a mídia que trafega por uma `Porta`. Seu atributo `mimeType` identifica o tipo de mídia, e o mapa `propriedades` guarda pares chave-valor que variam conforme esse tipo, como `resolucao` e `duracao` para um vídeo. A relação é uma composição `1 — 1`: toda porta descreve a mídia que por ela passa, e essa descrição não existe desvinculada da porta.

4.4 GRAFO DE TAREFAS

O grupo Grafo de Tarefas corresponde ao WDD: a representação do *pipeline* já instanciado e pronto para execução, que o *WorkflowManager* monta a partir da descrição enviada pelo cliente. Suas classes são *GrafoDeTarefas*, *Tarefa*, *Source* e *Sink*, mostradas na Figura 4.4. A relação com o grupo Função aparece dentro dele, na *Tarefa*: retomando a decisão de modelagem da Seção 4.1, cada tarefa é a instância em execução de uma *Funcao*. O grafo é, então, o conjunto dessas instâncias amarradas pelas conexões e pelas fronteiras de entrada e saída.

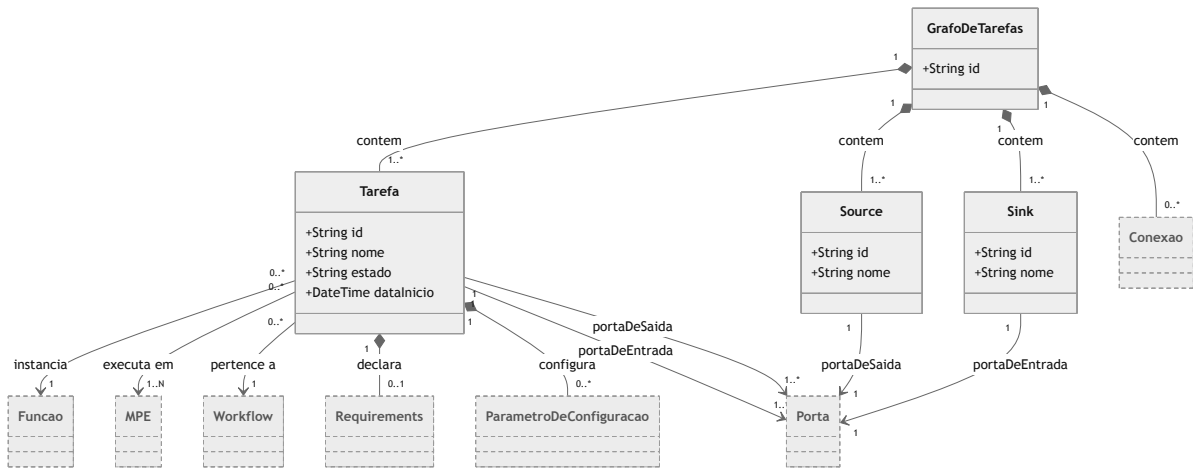


Figura 4.4: Recorte do modelo: o grupo Grafo de Tarefas (WDD).

A classe *GrafoDeTarefas* é a agregadora do grupo: reúne as *Tarefa* (1 — 1..*), as fronteiras *Source* e *Sink* (1 — 1..* cada) e as *Conexao* (1 — 0..*) que as ligam. É ela que dá unidade ao grafo dirigido, o conjunto de nós e arestas que define a ordem em que o processamento ocorre. Seus atributos resumem-se a um *id*, pois seu papel é estrutural: agregar os elementos do *pipeline*, não descrevê-los.

A classe *Tarefa* é o nó de processamento e a peça central do grupo. Cada *Tarefa* instancia uma *Funcao*, trazendo a definição estática e reutilizável para uma execução concreta: é o lado da instância no par *template*-instância discutido na Seção 4.1. Por isso, é na tarefa que vivem os dados próprios de uma execução: o atributo *estado* acompanha seu ciclo de vida (*pendente* ao ser criada, *executando* quando em curso e *concluída* ou *falha* ao terminar, sendo estes dois últimos estados terminais), e *dataInicio* registra quando a execução começou. A tarefa executa sobre um ou mais MPE, a entidade de

processamento detalhada adiante na seção *Processamento e Infraestrutura*. Ela também declara seus próprios `Requirements` e `ParametroDeConfiguracao`: do lado da tarefa, o parâmetro carrega o valor efetivo da execução, eventualmente sobrescrevendo o valor padrão herdado da função, detalhe retomado na seção *Configuração e Requisitos*.

Por fim, `Source` e `Sink` são as fronteiras do grafo: marcam por onde a mídia entra (`Source`) e por onde sai (`Sink`). Cada um expõe exatamente uma `Porta` (de saída no `Source`, de entrada no `Sink`), pela qual se conecta ao restante do *pipeline*. São nós simples, com atributos `id` e `nome`, cuja função é demarcar onde o fluxo de dados começa e termina.

4.5 CONFIGURAÇÃO E REQUISITOS

O grupo *Configuração e Requisitos* difere de todos os anteriores: ele não descreve um nó do grafo, mas um aspecto compartilhado por `Funcao` e `Tarefa`. Suas classes (`ParametroDeConfiguracao`, `Requirements` e `HardwareRequisito`) são declaradas tanto pela função quanto pela tarefa, e é justamente esse paralelo que justifica reuni-las em um grupo próprio, em vez de duplicá-las nos dois lados. A Figura 4.5 mostra essa estrutura transversal. O paralelo segue a separação entre definição e execução que atravessa o modelo: o que a `Funcao` declara, no FDD, são os valores padrão; o que a `Tarefa` declara, no WDD, são os valores efetivos daquela execução.

A classe `ParametroDeConfiguracao` representa um parâmetro ajustável de uma função ou tarefa. Seus atributos a identificam e descrevem (`id`, `nome`, `descricao`), enquanto o `valor` guarda o ajuste em si, de tipo livre, interpretado conforme o `tipoDeDado`. É nesse `valor` que se concretiza o mecanismo de sobrescrita antecipado na seção anterior: do lado da `Funcao`, ele carrega o valor padrão do parâmetro; do lado da `Tarefa`, o valor efetivo escolhido pelo cliente no WDD. Na ausência de uma escolha explícita na tarefa, prevalece o padrão herdado da função.

As classes `Requirements` e `HardwareRequisito` representam os recursos exigidos para a execução. A `Requirements` reúne exigências de desempenho (`atrasoMaximo` e `vazaoMinima`), ao passo que a `HardwareRequisito` detalha, item a item, as exigências de

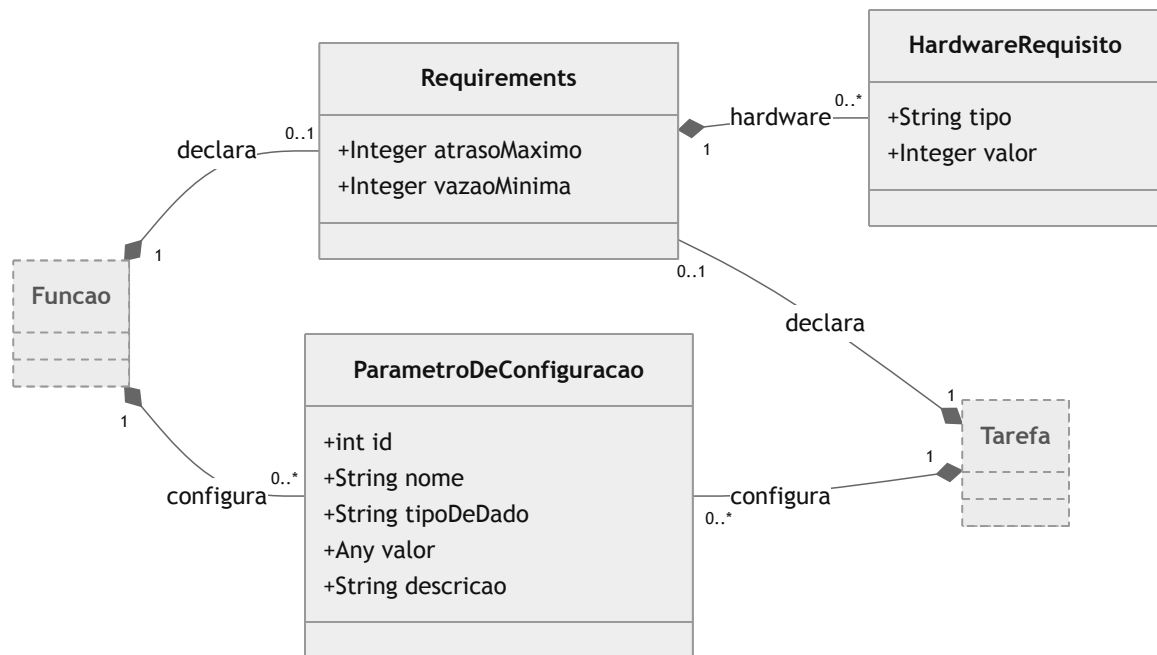


Figura 4.5: Recorte do modelo: o grupo Configuração e Requisitos, compartilhado por função e tarefa.

infraestrutura, com um **tipo** (como *vcpu*, *vgpu*, *ram* ou *disk*) e um **valor**; a composição 1 — 0..* permite que um mesmo conjunto de requisitos enumere vários itens de hardware. Vale a mesma lógica padrão-efetivo: a função declara os recursos de que necessita em tese, e a tarefa, os que exige naquela execução concreta.

4.6 PROCESSAMENTO E INFRAESTRUTURA

O grupo Processamento e Infraestrutura representa a entidade de processamento do NBMP: o MPE (*Media Processing Entity*), o ambiente sobre o qual as tarefas efetivamente executam. Suas classes são **MPE**, **Capabilities** e **RecursoDisponivel**, mostradas na Figura 4.6. Aqui se aplica novamente a decisão de modelagem da Seção 4.1: o MPE é mantido mínimo, e toda a descrição dos seus recursos vive na classe **Capabilities**, mais uma ocorrência do par entidade-vs-descrição que atravessa o modelo.

A classe **MPE** tem como único atributo um **id**: é o nó de processamento em si, propositalmente enxuto. Quem o descreve é a **Capabilities**, cujos atributos (**placement**, **location**) situam onde o MPE está e como é posicionado; a composição 1 — 1 indica que todo MPE tem exatamente uma descrição de capacidades. Além de localizá-lo, a

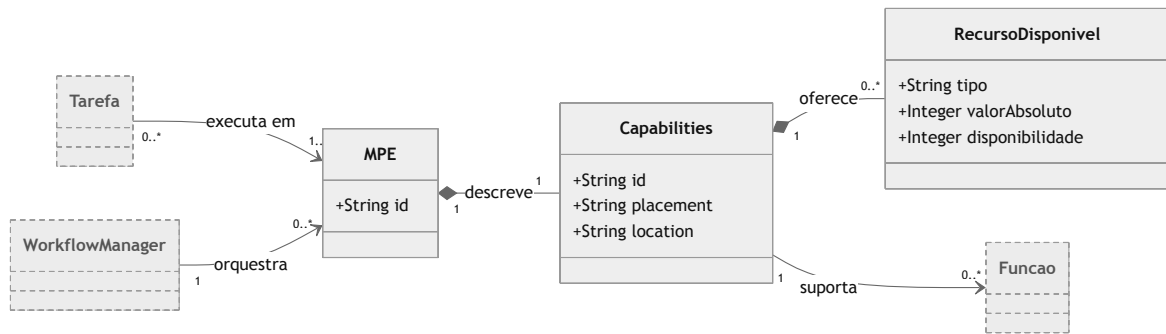


Figura 4.6: Recorte do modelo: o grupo Processamento e Infraestrutura (MPE).

Capabilities declara, pela relação **suporta**, quais funções aquele MPE é capaz de executar ($0..* \text{ Funcao}$), informação que permite ao orquestrador decidir onde cada tarefa pode rodar.

Por fim, a classe **RecursoDisponivel** enumera os recursos que a **Capabilities** oferece, item a item: o **tipo** (como *vcpu*, *vgpu*, *ram*, *disk* ou *power*), o **valorAbsoluto** e a **disponibilidade**, esta expressa em porcentagem. A composição $1 \text{ — } 0..*$ permite descrever vários recursos por MPE. Vale notar a simetria com a **HardwareRequisito** da seção anterior: lá, a tarefa declara o que exige; aqui, o MPE declara o que dispõe. É o casamento entre essa demanda e essa oferta que fundamenta a alocação de tarefas a MPEs.

4.7 WORKFLOW E ORQUESTRAÇÃO

O grupo Workflow e Orquestração é a camada de topo do modelo: aquela que amarra todos os demais e coloca a arquitetura em funcionamento. Suas classes são **Workflow** e **WorkflowManager**, mostradas na Figura 4.7. É por aqui que o sistema recebe a descrição enviada pelo cliente e a transforma em execução, acionando os grupos já apresentados: monta o **GrafoDeTarefas**, aloca as tarefas aos MPE e conduz todo o ciclo de vida. Por isso, esta seção fecha a apresentação do modelo.

A classe **Workflow** representa uma execução como um todo. Seus atributos (**id**, **nome**, **descricao**, **estado**) a identificam e registram a situação corrente da execução, e ela contém o **GrafoDeTarefas** que será executado (composição $1 \text{ — } 0..1$). É também o

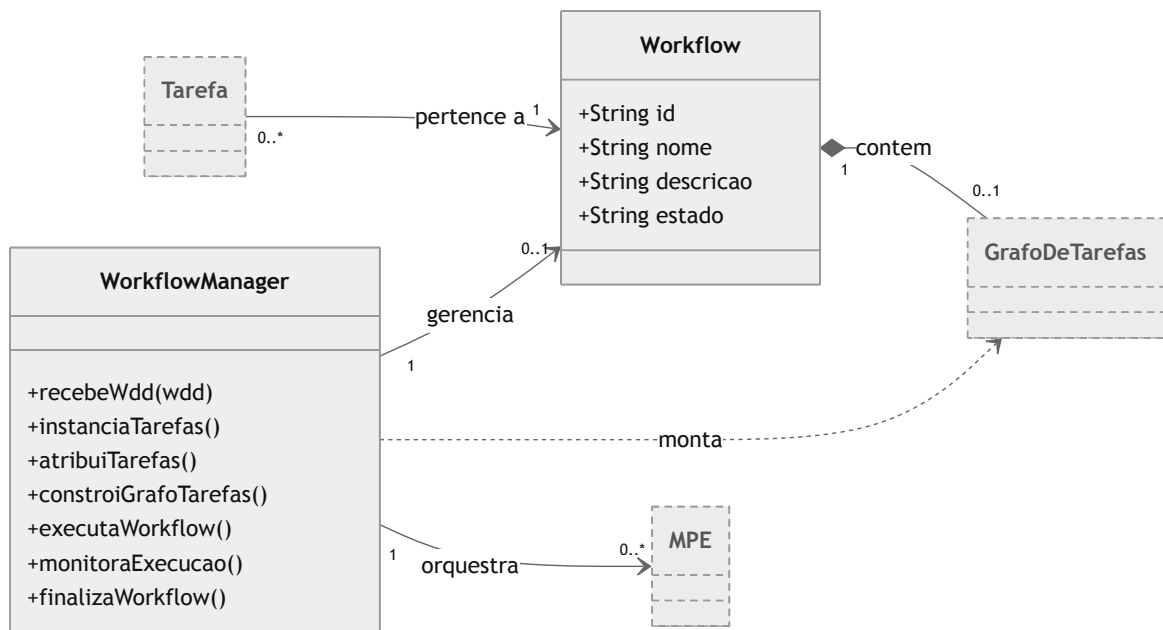


Figura 4.7: Recorte do modelo: o grupo Workflow e Orquestração.

agregador ao qual as tarefas pertencem: cada **Tarefa** está vinculada a um único **Workflow**. O atributo **estado** acompanha o ciclo de vida da execução por meio de uma máquina de estados simplificada em relação à definida pelo NBMP. O modelo fixa os estados possíveis, mas deixa deliberadamente em aberto como o estado das tarefas se propaga para o estado do workflow: a regra concreta que decide, a partir das tarefas, quando o workflow conclui ou falha é uma questão de implementação, tratada no capítulo seguinte.

A classe **WorkflowManager** é o orquestrador, a única classe do modelo definida por comportamento, e não por atributos. Suas operações descrevem o ciclo de vida de um *workflow*: `recebeWdd` recebe a descrição do cliente; `instanciaTarefas` e `constroiGrafoTarefas` traduzem essa descrição no grafo executável; `atribuiTarefas` aloca cada tarefa a um MPE; `executaWorkflow`, `monitoraExecucao` e `finalizaWorkflow` conduzem e encerram a execução. Suas relações fecham o modelo: o **WorkflowManager** gerencia o **Workflow**, monta o **GrafoDeTarefas** e orquestra os MPE. É nele que reside a dinâmica da arquitetura: enquanto os demais grupos descrevem a estrutura estática do sistema, o **WorkflowManager** é o agente ativo que a coloca em movimento.

5 IMPLEMENTAÇÃO

O modelo apresentado no capítulo anterior não é apenas um diagrama: ele é executável. Para demonstrá-lo, este capítulo descreve uma prova de conceito que materializa a arquitetura em código: um orquestrador genérico que lê os descritores do NBMP e executa um *pipeline* de processamento de mídia a partir deles. A premissa que orienta toda a implementação é direta: o comportamento de cada *pipeline* não está embutido no código, mas declarado nos descritores; trocar o descritor troca o *pipeline*, sem alterar uma linha de código. É essa mesma premissa que permite tratar funções de IA como componentes reutilizáveis entre *pipelines*: por viverem nos descritores, e não no código, podem ser reaproveitadas apenas alterando a descrição do *workflow*.

Essa premissa separa a implementação em dois comportamentos distintos. De um lado, a construção de um *pipeline* específico (quais funções participam, como se conectam e com quais parâmetros) vive inteiramente nos descritores JSON (WDD e FDD), montados pelo usuário. De outro, a execução desse *pipeline* (instanciar as tarefas, ordená-las, subir os *containers* e acompanhar seus estados) vive no código do orquestrador, que permanece o mesmo independentemente do *pipeline* descrito. É essa separação que sustenta a generalidade da arquitetura: um único motor executa qualquer *workflow* que o modelo seja capaz de descrever.

Este capítulo concentra-se nesse motor genérico. As seções seguintes apresentam as tecnologias e a organização do código, o mapeamento do modelo de classes em código, o papel dos descritores em tempo de execução, o fluxo de uma execução de ponta a ponta, a máquina de estados que governa tarefas e *workflows* e as fronteiras desta versão. A demonstração do motor em funcionamento (os dois casos de uso que o exercitam e os resultados obtidos) é o objeto do capítulo seguinte.

5.1 TECNOLOGIAS E ORGANIZAÇÃO

O orquestrador foi implementado em Python 3.11, utilizando o Docker SDK *for Python* para comandar a execução dos *containers*. A escolha do Docker como *runtime* decorre diretamente do modelo: cada função do diagrama corresponde a uma *Imagem*, e a

relação *Funcao* 1 — 1 *Imagem* materializa-se como uma imagem Docker por função. Assim, executar uma tarefa é subir um *container* a partir da imagem da função que ela instancia. Quanto ao processamento de mídia em si, vale o princípio de reúso: os filtros são construídos sobre bibliotecas consolidadas de visão computacional e processamento de mídia e sobre modelos pré-treinados, sem treino de modelos próprios. Essa decisão mantém o foco do trabalho na arquitetura, e não nos algoritmos de visão. As bibliotecas concretas de cada função são apresentadas no próximo capítulo, junto dos casos de uso que as empregam.

O código organiza-se em três frentes, que espelham a separação entre construção e execução discutida na seção anterior, conforme a Figura 5.1. A pasta *orquestrador/* abriga o *engine* genérico: o *WorkflowManager* e o maquinário que opera sobre o modelo. A pasta *descritores/* reúne os documentos WDD e FDD, em JSON, que descrevem os *pipelines*. E a pasta *funcoes/* contém a implementação (o filtro) de cada função, cada uma acompanhada de seu próprio *Dockerfile*, que define a imagem correspondente. As duas primeiras frentes são detalhadas nas próximas seções; a terceira, por ser específica de cada caso de uso, é retomada no capítulo seguinte.

<i>implementacao/</i>	
<i>orquestrador/</i>	engine genérico (<i>WorkflowManager</i> + maquinário)
<i>descritores/</i>	descritores WDD e FDD (JSON)
<i>funcoes/</i>	filtro de cada função (+ <i>Dockerfile</i> por função)
<i>midia/</i>	mídia de entrada e artefatos de saída

Figura 5.1: Organização do repositório da implementação em três frentes principais.

5.2 DO MODELO AO CÓDIGO

A implementação traduz o modelo de classes em três categorias de código, cuja distinção é essencial para compreendê-la. A primeira é o modelo: as classes do capítulo anterior, materializadas como estruturas de dados no pacote *orquestrador/modelo/*. A segunda é o maquinário: os “verbos” do *engine* (o *parser*, o *executor*, o *workflow_manager* e o ponto de entrada *main*) que operam sobre essas classes. A terceira são os filtros: o algoritmo que roda dentro de cada *container*, isolado do orquestrador. Em outras palavras, as classes do modelo guardam os dados; o comportamento vive no maquinário e nos filtros.

A Tabela 5.1 mostra onde cada um dos seis grupos do modelo apresentado no Capítulo 4 se materializa no código. Vale notar como o grupo Workflow e Orquestração se reparte entre o modelo e o maquinário: o `WorkflowManager`, definido por comportamento, é maquinário, enquanto o `Workflow` que ele gerencia pertence ao modelo.

Tabela 5.1: Materialização dos seis grupos do modelo (Capítulo 4) no código do orquestrador.

Grupo (Capítulo 4)	Onde vive no código
Função	<code>modelo/funcao.py</code>
Conectores do Grafo	<code>modelo/porta.py</code>
Grafo de Tarefas	<code>modelo/workflow.py</code> (GrafoDeTarefas, Tarefa); Source e Sink como Porta
Configuração e Requisitos	<code>modelo/requisitos.py</code> e <code>modelo/funcao.py</code> (ParametroDeConfiguracao)
Processamento	e <code>modelo/mpe.py</code>
Infraestrutura	
Workflow e Orquestração	<code>workflow_manager.py</code> (maquinário) e <code>modelo/workflow.py</code> (Workflow)

O maquinário articula-se em torno de duas traduções complementares: as pontes entre o mundo dos descritores e o da execução. A primeira é o ***parser*** (`parser_descritores.py`), que converte os descritores JSON nas classes do modelo: o texto vira objetos. A segunda é o ***executor*** (`executor_docker.py`), que converte cada tarefa do modelo em um *container* em execução: o objeto vira processo. A criação e a execução dos *containers* ficam concentradas no *executor*, que encapsula o trato com o Docker; ao orquestrador cabe acioná-lo para cada tarefa e reagir ao resultado.

5.3 OS DESCRITORES EM TEMPO DE EXECUÇÃO

Os descritores são a fonte da verdade em tempo de execução: é a leitura deles que constrói o *pipeline*. Há dois tipos, fiéis ao NBMP. O FDD (*Function Description Document*)

descreve uma função isolada e reutilizável (sua *imagem*, as portas de entrada e saída e os parâmetros), independentemente de qualquer *pipeline*; a Figura 5.2 traz um exemplo esquemático. O WDD (*Workflow Description Document*) monta o *pipeline* específico, no Modo A: referencia as funções participantes, declara o *source* e o *sink* e traz o *connection-map* explícito, em que cada item *origem* $\rightarrow$ *destino* dá origem a uma *Conexao*, e é desse mapa que o *engine* deriva o grafo de tarefas (Figura 5.3). A divisão reproduz a do modelo: o FDD é o bloco reutilizável (a *Funcao*), e o WDD é a montagem concreta (o *GrafoDeTarefas*).

```
{ "funcao": {
  "id": "funcao-a",
  "imagem": { "url": "registro/funcao-a:latest", "os": "linux" },
  "input-ports": [ { "nome": "entrada", "mimeType": "video/mp4" } ],
  "output-ports": [ { "nome": "saida", "mimeType": "application/json" } ],
  "parametros": [ { "nome": "limiar", "tipoDeDado": "int", "valor": 30 } ]
} }
```

Figura 5.2: Trecho esquemático de um FDD: a descrição de uma função isolada.

```
{ "workflow": {
  "funcoes": [
    { "ref": "funcao-a", "fdd": "fdd-funcao-a.json",
      "parametros": { "limiar": 10 } },
    { "ref": "funcao-b", "fdd": "fdd-funcao-b.json" }
  ],
  "source": { "id": "src",
    "porta-saida": { "nome": "entrada",
      "cachingServerUrl": "midia/entrada/video.mp4" } },
  "sink": { "id": "snk",
    "porta-entrada": { "nome": "resultado",
      "cachingServerUrl": "midia/saida/resultado.json" } },
  "connection-map": [
    { "origem": "src:entrada", "destino": "funcao-a:entrada" },
    { "origem": "funcao-a:saida", "destino": "funcao-b:entrada" },
    { "origem": "funcao-b:saida", "destino": "snk:resultado" }
  ]
} }
```

Figura 5.3: Trecho esquemático de um WDD: a montagem do *pipeline* no Modo A.

As chaves desses JSON são as do padrão, e cabe ao *parser* materializá-las nas classes do Capítulo 4. O mapeamento é direto, resumido na Tabela 5.2. É essa a tradução concreta que a primeira ponte da Seção 5.2 executa: o texto do descritor torna-se o grafo de objetos que o restante do *engine* manipula.

Tabela 5.2: Mapeamento das chaves dos descritores (vocabulário do padrão) nas classes do modelo.

Chave no descritor (JSON)	Classe no modelo
input-ports / output-ports	Porta
contentType	MidiaMetadata
imagem	Imagem
parametros	ParametroDeConfiguracao
connection-map (cada item)	Conexao

Dois comportamentos só ganham forma em tempo de execução. O primeiro é a distinção entre valor padrão e valor efetivo de um parâmetro, modelada na Seção 4.5: o *parser* lê o **valor** declarado no FDD como padrão e o substitui pelo *override* do WDD quando este existe. Sem *override*, prevalece o padrão. Nos trechos das figuras, a **funcao-a** tem **limiar** igual a 30 no FDD, sobrescrito para 10 no WDD. O segundo é a resolução de caminhos: os caminhos do **source** e do **sink** são declarados no WDD (no atributo **cachingServerUrl**), mas os caminhos intermediários (por onde uma função entrega dados à seguinte) não aparecem em descritor algum. É o *parser* que os deriva e copia o caminho da porta de origem para a porta de destino de cada conexão, garantindo por construção que cada tarefa leia exatamente o arquivo que a anterior escreveu.

5.4 O FLUXO DE EXECUÇÃO

Uma execução começa com o comando `main --wdd <arquivo>`, e o **WorkflowManager** a coordena de ponta a ponta, delegando às duas pontes apresentadas na Seção 5.2. O percurso completo desdobra-se em cinco passos:

1. O *parser* lê o WDD e os FDDs, materializa o modelo, resolve os caminhos das portas e deriva as dependências entre as tarefas a partir do **connection-map**.
2. O **WorkflowManager** pede ao **GrafoDeTarefas** a ordem topológica: aquela em que cada tarefa aparece só depois de todas as que a precedem no grafo.

3. Para cada tarefa, nessa ordem, o *executor* monta a Imagem, o comando (`--in/--out/--param`) e o volume compartilhado, e sobe um container distinto, o MPE daquela tarefa.
4. Dentro do container, executa-se a implementação da função: ela lê suas entradas do volume, processa e escreve as saídas de volta nele.
5. O *exit code* do container sobe pelo SDK; o *WorkflowManager* transiciona o estado da tarefa e, ao final, deriva o estado do *Workflow*.

Duas propriedades do fluxo merecem destaque. A primeira é que a execução é sequencial e segue a ordem topológica: uma tarefa só começa após o término de todas as que a precedem no grafo, um arranjo adequado a um *pipeline* encadeado. A segunda é o desacoplamento entre as tarefas: cada uma apenas lê suas entradas e escreve suas saídas, sem conhecer as demais. A comunicação entre elas se dá inteiramente pelo volume compartilhado, em que o arquivo de saída de uma tarefa é a entrada da seguinte, conforme as conexões do grafo. As transições de estado mencionadas no quinto passo são governadas pela máquina de estados, detalhada na próxima seção.

5.5 MÁQUINA DE ESTADOS

A execução de cada *Tarefa* é governada por uma máquina de estados simples, ilustrada na Figura 5.4. Uma tarefa nasce **pendente**; quando o *executor* a inicia, passa a **executando**; e, ao terminar, chega a **concluída**, se bem-sucedida, ou a **falha**, se o container retorna erro. Os estados **concluída** e **falha** são terminais: não há *reset* nem *destroy*, pois a prova de conceito executa cada *workflow* uma única vez. Trata-se de uma simplificação consciente da máquina mais rica prevista pelo NBMP (que inclui estados como *idle*, *running*, *in-error* e *destroyed*), adequada ao caráter pontual da execução.

O *Workflow* reaproveita a mesma máquina, mas seu estado é derivado do conjunto de tarefas: ele se conclui quando todas as tarefas concluem e entra em **falha** assim que qualquer uma falha, interrompendo a execução: as tarefas seguintes não chegam a iniciar.

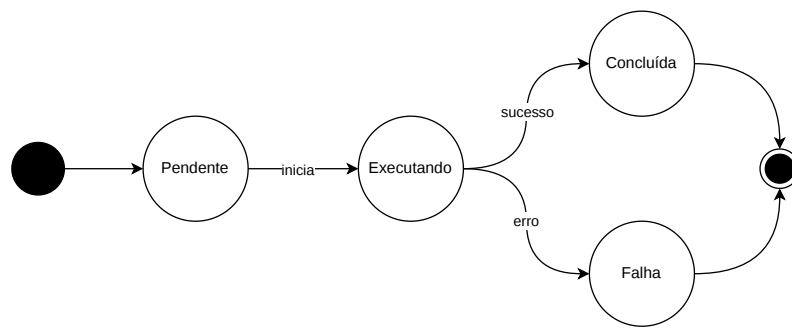


Figura 5.4: Máquina de estados que governa a execução de cada **Tarefa**: de **pendente** a **executando** e, ao término, aos estados terminais **concluída** ou **falha**.

É justamente a regra que a Seção 4.7 havia deixado a cargo da implementação: como o estado das tarefas se propaga para o estado do *workflow*.

5.6 FRONTEIRAS DESTA VERSÃO

As reduções desta versão do *engine* são escolhas de escopo, e não limitações do modelo. Três delas merecem registro. A primeira é o transporte por volume compartilhado: as tarefas trocam dados por meio de arquivos em um volume local ao *host*, o mecanismo mais simples para a versão atual; uma evolução para HTTP ou *streaming* ficaria concentrada no *executor*. A segunda é a execução sequencial e encadeada: o fluxo apresentado atende a um *pipeline* linear, ao passo que um *workflow* com ramos paralelos independentes exigiria executá-los de forma concorrente e isolar uma eventual falha ao ramo afetado. A terceira é a restrição a um único *host*: os containers executam todos na mesma máquina, e não como nós de um *cluster* distribuído real.

Nenhuma dessas reduções enfraquece a arquitetura: elas delimitam o que esta versão do motor faz, não o que o modelo permite. O próximo capítulo evidencia essa distinção ao mostrar o mesmo *engine*, sem alterações em seu núcleo, executando uma única tarefa como vários containers concorrentes: um paralelismo *dentro* de uma tarefa, que convive com a mesma ordem sequencial entre as tarefas descrita aqui. Os caminhos de evolução mais amplos (paralelismo entre ramos independentes do grafo, distribuição real entre máquinas e transporte por *streaming*) são retomados como trabalhos futuros no Capítulo 7.

6 CASOS DE USO E RESULTADOS

O motor genérico descrito no capítulo anterior só se justifica se for capaz de executar *pipelines* reais. Para demonstrá-lo, dois casos de uso de domínios distintos foram construídos e executados sobre o mesmo orquestrador. O primeiro (reconhecimento facial em vídeo) valida o fluxo básico de funções encadeadas, em que a fonte de mídia alimenta o vídeo a todas elas. O segundo (leitura de números em *displays* de sete segmentos) reusa esse mesmo motor, mas exige mais dele: paralelismo *dentro* de uma tarefa e seleção do recurso de processamento que a executa. O critério central que os dois casos verificam é único: o mesmo orquestrador executa ambos sem nenhum desvio condicional por caso de uso: toda a diferença entre eles vive nos descritores, não no código. Em ambos os casos, as funções centrais são baseadas em IA (reconhecimento facial e leitura óptica de dígitos), justamente a classe de função que a arquitetura visa integrar e reusar entre *pipelines*.

Os resultados são observados pela execução de ponta a ponta de cada *workflow* sobre vídeos de demonstração: considera-se bem-sucedida a execução em que o *workflow* conclui e a saída é coerente com o *pipeline* descrito. Vale uma ressalva de escopo: a qualidade dos algoritmos de visão (detecção facial, reconhecimento óptico) não é o objeto de avaliação deste trabalho. O que se avalia é o comportamento do motor; os algoritmos entram apenas como conteúdo das funções, construídos sobre bibliotecas consolidadas.

6.1 CASO 1: RECONHECIMENTO FACIAL

O Caso 1 ataca um problema concreto de análise de vídeo: detectar, identificar e rotular as pessoas que aparecem ao longo de um vídeo. O *workflow* que o resolve está ilustrado na Figura 6.1: a fonte de mídia distribui o vídeo original para as três funções de processamento: F1 precisa dele para localizar os rostos, F2 para codificar cada rosto a partir dos *frames*, e F3 para desenhar as anotações sobre a imagem original. Cada seta do grafo transporta, pelo volume compartilhado, o artefato que uma função entrega à seguinte, conforme o modelo de *portas e conexões* do Capítulo 4.

As três funções dividem o trabalho em etapas encadeadas. **F1** amostra os *frames* do vídeo (um a cada `intervalo_frames`) e detecta os rostos presentes em cada um, com

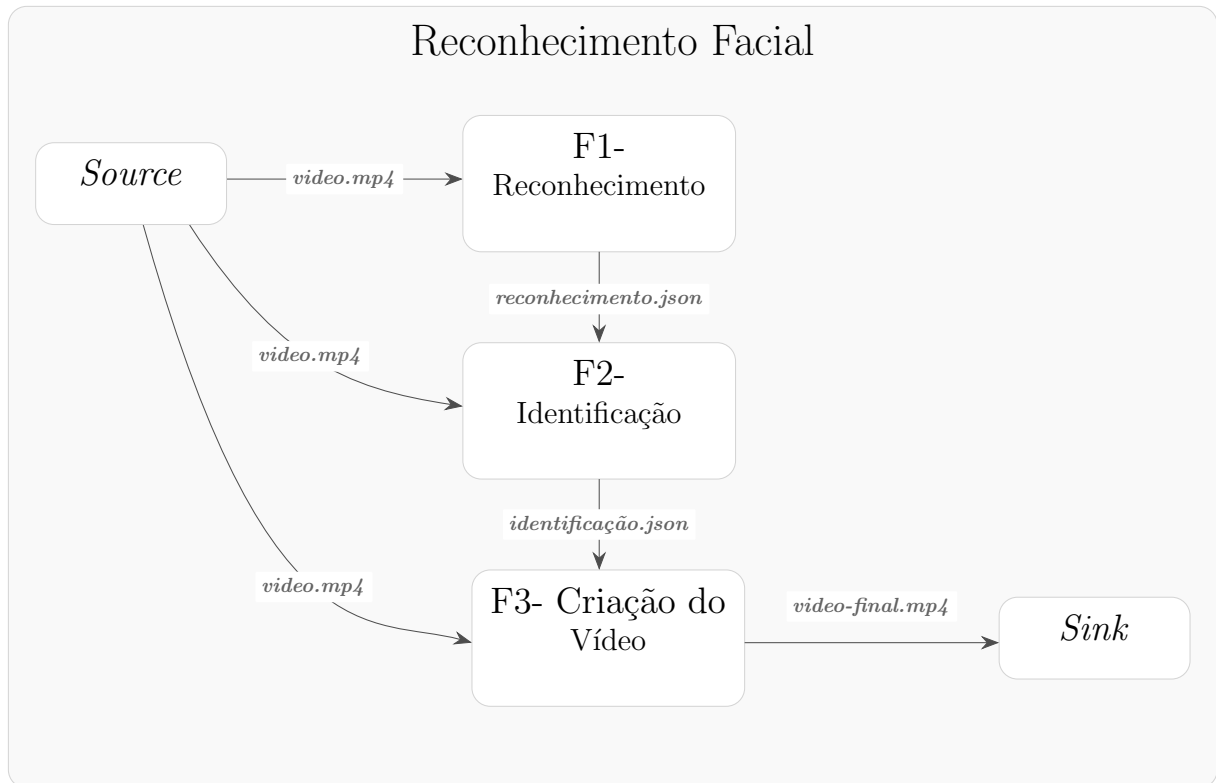


Figura 6.1: *Pipeline* do Caso 1: reconhecimento facial. A fonte de mídia distribui o vídeo para as três funções; cada seta indica o artefato trafegado pelo volume compartilhado.

OpenCV (Bradski, 2000) e a biblioteca *face_recognition* (Geitgey, 2017), entregando uma lista com a região que cada rosto ocupa na imagem. **F2** percorre essas detecções, calcula o vetor de características (*encoding*) de cada rosto e os agrupa por similaridade, atribuindo a cada grupo uma identidade (**person_N**). **F3** desenha sobre o vídeo original as caixas e os rótulos de cada identidade e codifica o resultado com *FFmpeg* (FFmpeg Project, 2024), produzindo o vídeo anotado. Cada função tem uma imagem Docker e, a cada execução, o *Workflow Manager* cria uma tarefa que utiliza essa imagem, exatamente como descrito no Capítulo 5.

O *workflow* foi executado sobre um vídeo de demonstração com oito pessoas. Para isso, o WDD redefiniu o *ParametroDeConfiguracao* *intervalo_frames* de F1, de seu valor padrão (30) para 10, um exemplo concreto da resolução *padrão* $\times$ *efetivo* descrita no Capítulo 5, em que o valor declarado no WDD prevalece sobre o *default* do FDD. A execução percorreu toda a cadeia e terminou com o vídeo anotado e o *workflow* no estado *concluido*; a Figura 6.2 mostra um *frame* do resultado, com pessoas identificadas

e rotuladas. Mais do que a saída em si, o que o caso evidencia é o comportamento do motor: a redefinição de um parâmetro pelo descritor alterou a execução sem qualquer modificação no código do orquestrador.

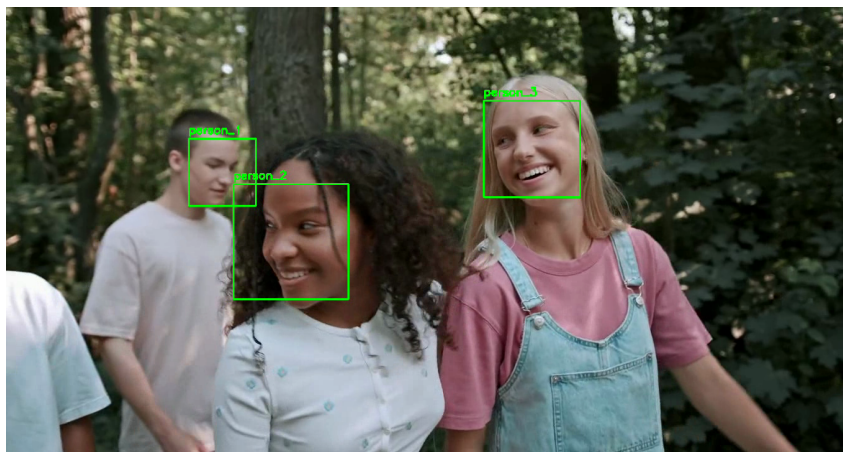


Figura 6.2: *Frame* do vídeo de saída do Caso 1, com três pessoas identificadas e rotuladas.

6.2 CASO 2: NÚMEROS EM *DISPLAY* DE SETE SEGMENTOS

O Caso 2 resolve um problema de outra natureza: ler os números exibidos em um *display* de sete segmentos ao longo de um vídeo. O *workflow* que o resolve tem quatro funções: **ExtratorDeFrames**, **DistribuiMidia**, **F4** (reconhecimento de sete segmentos) e **AgregaMidia**, encadeadas como mostra a Figura 6.3. O que torna este caso relevante não é o domínio, mas o fato de ele rodar no mesmo *Workflow Manager* do Caso 1, sem que o orquestrador contenha qualquer lógica específica para um caso ou outro, e ainda assim exercitar três capacidades do modelo que o Caso 1 não chegava a usar, examinadas nos parágrafos seguintes.

A primeira capacidade aparece já na divisão das funções. Três das quatro são funções-base, genéricas e sem qualquer traço do domínio “sete segmentos”: **ExtratorDeFrames** decompõe o vídeo em *frames* (com *OpenCV*), **DistribuiMidia** divide esses *frames* em *N* partições e **AgregaMidia** concatena e ordena os resultados parciais ao final. Por não dependerem do domínio, residem no **RepositorioDeFuncoes** e podem ser reaproveitadas por outros *workflows*. Apenas **F4** é específica: lê o número de cada *frame*, combinando *OpenCV* para isolar os segmentos acesos e a ferramenta **ssocr** (Auerswald, 2008) para

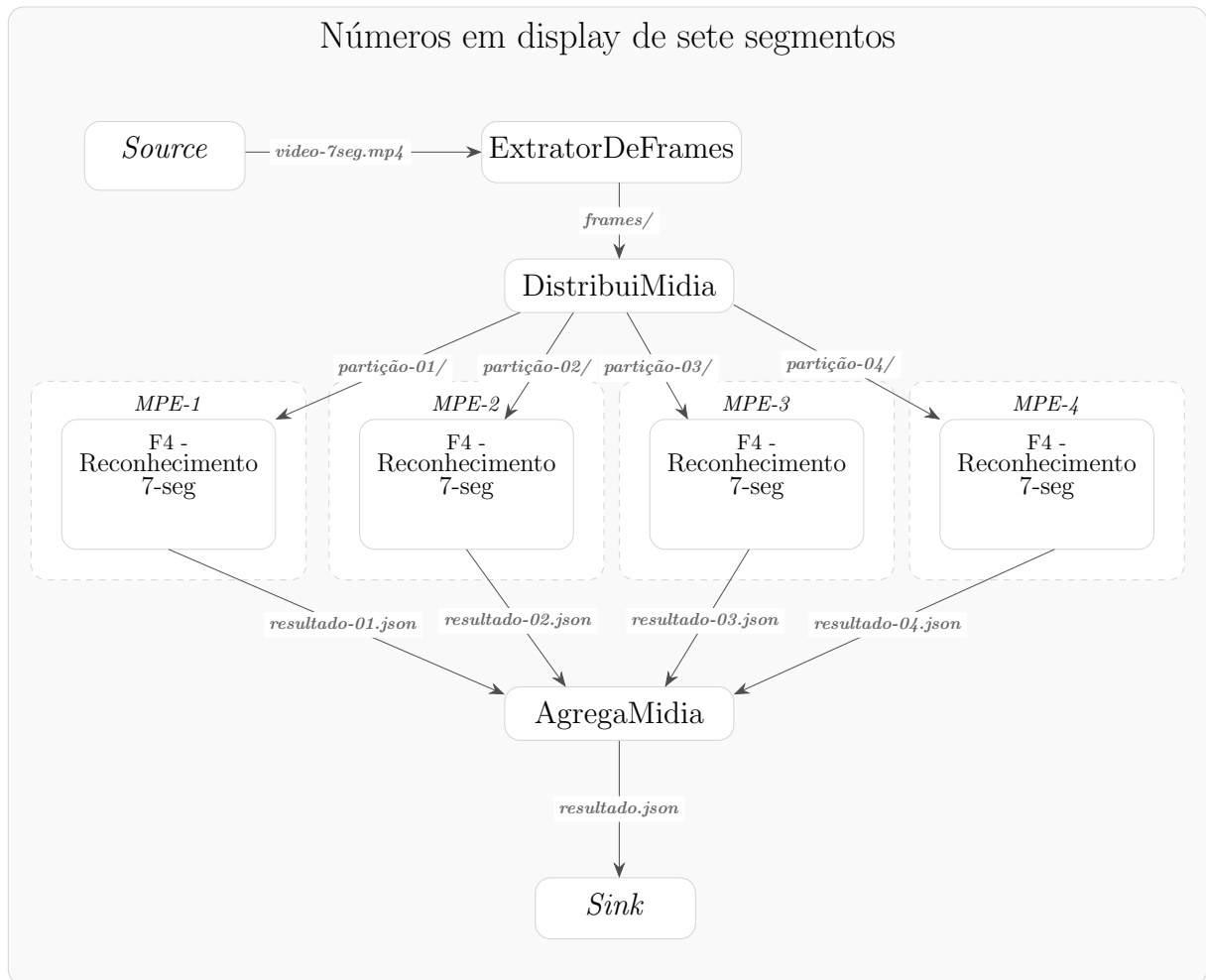


Figura 6.3: *Pipeline* do Caso 2: leitura de números em *display* de sete segmentos. A tarefa de F4 é distribuída em 4 MPEs concorrentes, e **Agregamidia** reúne os resultados parciais em um único **resultado.json**.

interpretar os dígitos.

A segunda capacidade é o paralelismo dentro de uma única tarefa: a cardinalidade **Tarefa 1 — 1..N MPE** do modelo (Capítulo 4) posta em prática. A tarefa de F4 não roda em um *container* só: ela é executada por N *containers* concorrentes, um para cada partição produzida por **DistribuiMidia**. O motor genérico expressa isso sem qualquer tratamento específico: a porta de entrada de F4 declara o atributo **consumo: scatter** e o executor, ao encontrá-lo, enumera os itens da entrada e dispara um *container* por item. Sem esse atributo, a tarefa rodaria em um único MPE, com a entrada inteira, e é exatamente esse o comportamento do Caso 1, que pode ser visto como o caso particular de um só item. Alterar **n_particoes** no WDD muda o número de MPEs sem alterar o código.

A terceira capacidade é a seleção do recurso de processamento que executa a tarefa, ativada apenas quando o FDD da função declara um bloco de **requirements**. No Caso 2, F4 exige um mínimo de *hardware* (núcleos de processamento e memória), e o *Workflow Manager* casa esses requisitos com as capacidades dos MPEs disponíveis, despachando cada partição somente a um MPE apto. O conjunto de MPEs disponíveis vem de um descritor de ambiente (**mpes-disponiveis.json**), que representa o retorno de um monitoramento da infraestrutura, o levantamento dos MPEs ativos e de suas capacidades. Esse provisionamento de recursos é uma das responsabilidades que o NBMP delega à implementação, assumida aqui pelo próprio orquestrador. Quando nenhum MPE atende aos requisitos, a tarefa não é despachada e o *workflow* não executa.

Na execução sobre um vídeo de demonstração, a tarefa de F4 foi distribuída em vários *containers* concorrentes, e o resultado obtido ao repartir o trabalho é idêntico ao obtido com uma única partição, evidência de que dividir o trabalho não altera a saída. As partições foram despachadas apenas aos MPEs aptos, e um conjunto formado só por MPEs insuficientes produziu, como esperado, a recusa de execução descrita acima. Por fim, vale registrar que incluir o Caso 2 não exigiu qualquer modificação no motor: o Caso 1 permanece executando sem alterações sobre o mesmo orquestrador, o que confirma que as três capacidades acima vivem nos descritores, e não em mudanças no *engine*. A

Figura 6.4 mostra um *frame* do vídeo de entrada, com o *display* de sete segmentos que F4 interpreta; a saída do *workflow* é um único `resultado.json`, montado por `AgregaMidia`, com a leitura de cada *frame* em ordem, conforme o trecho da Figura 6.5.



Figura 6.4: *Frame* do vídeo de entrada do Caso 2: o *display* de sete segmentos lido por F4.

```
{ "n_particoes": 4,
  "frames": [
    { "frame": 0, "deteccoes": [ { "bbox": [322, 20, 226, 164], "valor": "38" } ] },
    { "frame": 1, "deteccoes": [ { "bbox": [322, 20, 226, 164], "valor": "38" } ] },
    //...
  ] }
```

Figura 6.5: Trecho do `resultado.json` do Caso 2.

6.3 SÍNTESE DOS RESULTADOS

Os dois casos, em conjunto, sustentam o argumento central deste trabalho. Um mesmo orquestrador executou desde uma cadeia linear simples até um *pipeline* com funções-base reutilizáveis, paralelismo dentro da tarefa e seleção do recurso de processamento. Toda essa variação coube nos descritores, sem qualquer lógica específica por caso de uso no código. Isso confirma que a arquitetura proposta não se esgota no diagrama de classes do Capítulo 4: interpretada e materializada em código, ela é executável e genérica, como antecipava o Capítulo 5. Soma-se a isso a integração de funções baseadas em IA como funções NBMP comuns, reusadas pelo mesmo motor sem tratamento específico, contribuição parcial cujo desdobramento, assim como as reduções assumidas nesta versão e os caminhos para superá-las, é retomado na conclusão.

7 CONCLUSÃO E TRABALHOS FUTUROS

Partindo de um padrão extenso e essencialmente textual, este trabalho o traduziu em uma arquitetura de referência e a conduziu até a execução, estreitando a distância entre ler a norma e construir um sistema que a implemente. Do diagrama de classes que organiza as entidades do NBMP (Capítulo 4) ao motor que as executa a partir de descritores (Capítulo 5), a proposta foi conduzida até a prova final: dois *pipelines* de domínios distintos levados do início ao fim (Capítulo 6). Mais do que reorganizar a norma, o que se entrega é a evidência de que o modelo dela derivado se sustenta na prática.

A mesma base de código que recebe um vídeo e devolve rostos rotulados também lê números em um *display* e os transcreve, sem que nada nela precise distinguir um caso do outro; a distância entre os dois domínios mora por inteiro nos arquivos de descrição. É esse deslocamento (da lógica para os dados) que dá às funções de IA o lugar que motivou a pesquisa: deixam de nascer presas a um fluxo e passam a ser peças que se encaixam onde forem chamadas. Ainda assim, convém medir o alcance do que se afirma. O que se construiu é uma demonstração de viabilidade, executada em uma única máquina e em passos sequenciais, e não um sistema pronto para produção. É dessa fronteira que partem os trabalhos futuros reunidos na Seção 7.3.

7.1 CONTRIBUIÇÕES

As contribuições deste trabalho podem ser resumidas em quatro frentes:

- um modelo arquitetural derivado do NBMP, que traduz a norma em classes, relações e decisões de projeto, uma ponte compreensível entre o texto normativo e quem precisa implementá-lo;
- uma implementação de referência genérica: um orquestrador que executa qualquer *workflow* descrito por seus descritores, sem lógica dedicada a cada caso de uso;
- a validação empírica da arquitetura por dois casos de domínios distintos, levados de ponta a ponta sobre o mesmo motor;

- a integração de IA como função NBMP comum, mostrando ser possível acomodar funções de IA no modelo sem tratamento especial, com a classe `SoftwareCapability` reservando o espaço para formas mais ricas de descrevê-las.

7.2 LIMITAÇÕES DO MODELO

O Capítulo 5 já delimitou as fronteiras desta versão da implementação: o transporte por volume compartilhado, a execução sequencial e a restrição a um único *host*, todas escolhas de escopo do motor, e não do modelo. Esta seção trata de uma questão distinta e independente da implementação: que tipos de *workflow* a arquitetura, enquanto modelo, é capaz de descrever e processar.

O primeiro limite não está na arquitetura, mas no que a cerca. O modelo é um orquestrador genérico e agnóstico ao que cada função faz: ele não processa mídia, coordena funções. O espaço de *workflows* realizáveis é, portanto, delimitado antes de tudo pelo catálogo de funções à disposição; uma capacidade que nenhuma função existente ofereça não pode ser composta, por mais expressivo que o modelo seja.

Fixado o conjunto de funções, a própria arquitetura ainda restringe as topologias e os modos de construção que consegue expressar:

- **Construção apenas no Modo A:** o cliente precisa descrever o grafo completo do *workflow* por meio do `connection-map`; o modelo não descobre nem seleciona funções automaticamente a partir de um objetivo desejado, o que corresponderia ao Modo B, retomado entre os trabalhos futuros (Seção 7.3);
- **Grafos acíclicos:** o *workflow* é representado como um grafo direcionado acíclico (DAG), de modo que fluxos com ciclos ou realimentação estão fora do que o modelo expressa;
- **Ausência de funções compostas:** o modelo não prevê *function groups*, isto é, funções que encapsulam um sub-*workflow* inteiro; cada nó do grafo é uma função única;

- **Subconjunto de descritores:** apenas os descritores essenciais ao núcleo da arquitetura foram modelados; capacidades apoiadas em descritores omitidos, como eventos, variáveis, segurança, asserções, processamento em etapas e *client-assistance*, ficam fora do que o modelo é capaz de descrever.

7.3 TRABALHOS FUTUROS

Vários dos limites discutidos até aqui apontam, ao mesmo tempo, o caminho do que vem a seguir. O mais promissor diz respeito à classe **SoftwareCapability**: hoje apenas um texto que descreve o que cada função faz, ela foi criada para um papel maior. Enriquecê-la, seja ancorando suas capacidades em uma ontologia de domínio, seja representando-as como vetores para uma busca semântica, permitiria que o próprio orquestrador descobrisse e selecionasse funções a partir do que se deseja obter. É esse mecanismo que realizaria o Modo B de construção do *workflow*, no qual as funções de IA, pela sua variedade, mais teriam a ganhar.

Uma extensão complementar amplia a própria descrição das funções para abarcar mais cenários de IA. Além das capacidades (**SoftwareCapability**) e dos requisitos de *hardware* (**HardwareRequisito**) que o modelo já carrega, cada função poderia declarar também suas dependências: as bibliotecas, os modelos e demais artefatos de *software* de que precisa para ser executada. Descrever cada função de IA por esses três eixos complementares (o que ela faz, o que exige da infraestrutura e do que depende para rodar) daria à arquitetura condições de acomodar com segurança uma variedade maior de funções.

As demais frentes seguem as fronteiras técnicas registradas no Capítulo 5. A execução, hoje confinada a uma única máquina e a passos sequenciais, pode evoluir para a distribuição real das tarefas entre vários *hosts* e para o paralelismo entre ramos independentes do grafo, ampliando o que o *scatter* intratarefa já demonstrou. No transporte de dados, o volume compartilhado que liga as etapas pode dar lugar à comunicação por *streaming* ou HTTP entre os *containers*, aproximando a prova de conceito de cenários de produção. E a integração de IA, já funcional como função comum, pode amadurecer até um módulo padronizado mais completo, que contemple também o

treinamento e a validação de funções baseadas em *machine learning*.

Mais do que uma lista de pendências, esses caminhos reafirmam o ponto de partida deste trabalho: o de oferecer uma base executável, e não apenas descritiva, sobre a qual o processamento de mídia em rede baseado no NBMP, e as funções de IA que o povoam, possam continuar a evoluir.

REFERÊNCIAS

- Auerswald, E., 2008. Seven Segment Optical Character Recognition. <https://github.com/auerswal/ssocr>, acesso em: 22 jun. 2026.
- Bradski, G., 2000. The opencv library. Dr. Dobb's Journal: Software Tools for the Professional Programmer 25 (11), 120–123.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 2008. Pattern-Oriented Software Architecture: A System of Patterns, Volume 1. Vol. 1. John Wiley & Sons.
- FFmpeg Project, 2024. FFmpeg. <https://ffmpeg.org>, acesso em: 22 jun. 2026.
- Geitgey, A., 2017. face_recognition: Recognize and manipulate faces from Python. https://github.com/ageitgey/face_recognition, acesso em: 22 jun. 2026.
- Guedes, A., Artificial Intelligence Applications in TV Industry Workflow. In: Anais Estendidos do XXXI Simpósio Brasileiro de Sistemas Multimídia e Web, 2025. SBC, Porto Alegre, RS, Brasil, pp. 265–270. Disponível em: https://sol.sbc.org.br/index.php/webmedia_estendido/article/view/38182
- ISO/IEC, 2023. ISO/IEC 23090-8:2023 – Information technology – Coded representation of immersive media – Part 8: Network-based media processing. Standard, International Organization for Standardization, Geneva, Switzerland.
- Martín-Pérez, J., Cominardi, L., Bernardos, C. J., de la Oliva, A., Azcorra, A., 2019. Modeling Mobile Edge Computing Deployments for Low Latency Multimedia Services. IEEE Transactions on Broadcasting 65 (2), 464–474.
- Neugebauer, M., Nagare Media Engine: Towards Self-Adapting MPEG NBMP Multimedia Workflows. In: Proceedings of the 16th ACM Multimedia Systems Conference, 2025. pp. 263–269.
- Ramos-Chavez, R., Mekuria, R., Karagkioules, T., Griffioen, D., Wagenaar, A., Ogle, M., MPEG NBMP Testbed for Evaluation of Real-Time Distributed Media Processing

Workflows at Scale. In: Proceedings of the 12th ACM Multimedia Systems Conference, 2021. MMSys '21. pp. 173–185.

Xu, Y., Yin, J., Yang, Q., Yang, L., 2022. Media Production Using Cloud and Edge Computing: Recent Progress and NBMP-Based Implementation. IEEE Transactions on Broadcasting 68 (2), 545–558.