**FlowScript - Documentação Completa**

**Índice**

1. [Introdução](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#introdu%C3%A7%C3%A3o)
2. [Instalação e Dependências](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#instala%C3%A7%C3%A3o-e-depend%C3%AAncias)
3. [Sintaxe da Linguagem](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#sintaxe-da-linguagem)
4. [Compilação e Execução](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#compila%C3%A7%C3%A3o-e-execu%C3%A7%C3%A3o)
5. [Sistema de Plugins](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#sistema-de-plugins)
6. [Integração com Serviços](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#integra%C3%A7%C3%A3o-com-servi%C3%A7os)
7. [Tratamento de Erros](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#tratamento-de-erros)
8. [Monitoramento e Debugging](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#monitoramento-e-debugging)
9. [Exemplos Práticos](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#exemplos-pr%C3%A1ticos)
10. [API Reference](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#api-reference)
11. [Melhores Práticas](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#melhores-pr%C3%A1ticas)
12. [Troubleshooting](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#troubleshooting)

**Introdução**

**FlowScript** é uma linguagem domain-specific (DSL) projetada especificamente para criação de workflows baseados em eventos com integração nativa de IA. A linguagem combina sintaxe natural intuitiva com capacidades de execução assíncrona e otimização automática.

**Características Principais**

* **Sintaxe Natural**: Próxima à linguagem humana para facilitar compreensão por IA
* **Eventos Assíncronos**: Sistema de eventos de primeira classe
* **Plugins Dinâmicos**: Extensibilidade através de plugins personalizados
* **IA Integrada**: Otimização e análise automática por inteligência artificial
* **Error Handling**: Tratamento robusto de erros com estratégias de recuperação
* **Monitoramento**: Observabilidade completa de execução

**Casos de Uso**

* Aprovação de empréstimos e processos financeiros
* Onboarding de clientes
* Processamento de documentos
* Workflows de aprovação empresarial
* Automação de processos de negócio
* Pipelines de dados com IA

**Instalação e Dependências**

**Requisitos do Sistema**

* **Python**: 3.8 ou superior
* **Sistema Operacional**: Linux, macOS, Windows
* **Memória**: Mínimo 2GB RAM
* **Armazenamento**: 100MB de espaço livre

**Dependências Python**

# Dependências obrigatórias

pip install asyncio dataclasses typing

# Dependências opcionais (para funcionalidades avançadas)

pip install aiohttp # Para integração HTTP

pip install aioredis # Para cache Redis

pip install sqlalchemy # Para integração com banco de dados

pip install pydantic # Para validação de schemas

pip install prometheus-client # Para métricas

**Instalação do FlowScript**

# Clonar repositório

git clone https://github.com/yourorg/flowscript.git

cd flowscript

# Instalar dependências

pip install -r requirements.txt

# Instalação do pacote

pip install -e .

**Estrutura de Diretórios**

flowscript/

├── src/

│ ├── flowscript/

│ │ ├── lexer.py # Analisador léxico

│ │ ├── parser.py # Parser da linguagem

│ │ ├── runtime.py # Runtime de execução

│ │ ├── compiler.py # Compilador

│ │ ├── plugins/ # Plugins built-in

│ │ ├── templates/ # Templates de workflow

│ │ └── tools/ # Ferramentas de desenvolvimento

│ └── tests/ # Testes unitários

├── examples/ # Exemplos de uso

├── docs/ # Documentação adicional

├── requirements.txt # Dependências

└── setup.py # Configuração de instalação

**Verificação da Instalação**

# test\_installation.py

from flowscript import FlowScriptCompiler, FlowScriptRuntime

def test\_installation():

"""Testa se FlowScript foi instalado corretamente"""

# Teste básico de compilação

code = """

@workflow TestWorkflow {

@context {

message: String

}

@flow {

start {

notify message

}

}

}

"""

try:

compiler = FlowScriptCompiler()

workflow = compiler.compile(code)

print("✅ FlowScript instalado com sucesso!")

print(f"Workflow compilado: {workflow.name}")

return True

except Exception as e:

print(f"❌ Erro na instalação: {e}")

return False

if \_\_name\_\_ == "\_\_main\_\_":

test\_installation()

**Sintaxe da Linguagem**

**Estrutura Básica**

Todo workflow FlowScript segue a estrutura fundamental:

@workflow NomeDoWorkflow {

@context {

// Definição de variáveis globais

}

@triggers {

// Mapeamento de eventos para ações

}

@flow {

// Definição dos steps do workflow

}

@error\_handling {

// Tratamento de erros

}

@config {

// Configurações do workflow

}

}

**Seção @context**

Define variáveis e tipos disponíveis globalmente no workflow:

@context {

customer: Customer, // Objeto customer

application: LoanApplication, // Aplicação de empréstimo

score: Number, // Score numérico

approved: Boolean, // Flag booleano

documents: DocumentList, // Lista de documentos

metadata: JsonObject // Objeto JSON genérico

}

**Tipos Suportados:**

* String: Texto
* Number: Números inteiros ou decimais
* Boolean: true/false
* Duration: Tempo (30s, 5m, 2h)
* JsonObject: Objeto JSON genérico
* Array: Lista de elementos
* Tipos customizados (Customer, LoanApplication, etc.)

**Seção @triggers**

Mapeia eventos para steps iniciais:

@triggers {

application\_received => validate\_data,

credit\_check\_complete => decision\_point,

user\_input\_received => process\_input,

timer\_expired => cleanup\_workflow,

external\_api\_callback => handle\_callback

}

**Tipos de Eventos:**

* **Eventos de Sistema**: workflow\_started, timeout\_reached
* **Eventos Customizados**: Definidos pelo desenvolvedor
* **Eventos Externos**: Vindos de APIs, webhooks, etc.
* **Eventos Temporais**: Baseados em tempo ou cronogramas

**Seção @flow**

Define a lógica principal do workflow através de steps:

@flow {

step\_name {

// Ações sequenciais

action1

action2

// Condicionais

if condition => next\_step

else => alternative\_step

// Branches múltiplos

branch {

condition1 => action1,

condition2 => action2,

condition3 => action3

}

// Chamadas de serviço

call Service.method(params) -> result\_var

// Emissão de eventos

emit event\_name

// Aguardar eventos

wait\_for event\_name

}

}

**Condicionais**

**Condicionais Simples**

if score > 700 => approve\_automatically

else => manual\_review

**Condicionais Compostas**

if score > 700 && risk\_level == "low" => premium\_approval

else if score > 600 => standard\_approval

else => detailed\_review

**Branch Statements**

branch {

score >= 750 && risk == "low" => auto\_approve\_premium,

score >= 650 && risk != "high" => auto\_approve\_standard,

score >= 550 => manual\_review,

score < 550 => auto\_reject

}

**Operadores**

**Operadores de Comparação**

* ==: Igual
* !=: Diferente
* >: Maior que
* <: Menor que
* >=: Maior ou igual
* <=: Menor ou igual

**Operadores Lógicos**

* &&: E lógico
* ||: OU lógico
* !: NÃO lógico

**Operadores de Atribuição**

* ->: Atribui resultado de chamada
* =: Atribuição simples

**Chamadas de Serviço e Plugins**

// Chamada de serviço externo

call CreditBureau.getScore(customer.ssn) -> credit\_score

// Chamada de plugin

call EmailSender(customer.email, "Subject", "Body") -> email\_result

// Chamada com múltiplos parâmetros

call RiskAnalyzer.analyze(customer, application, credit\_score) -> risk\_assessment

// Chamada sem retorno

call NotificationService.send("Message", customer.phone)

**Ações Built-in**

// Verificações

check application.completeness

check customer.eligibility

// Aprovações/Rejeições

approve application

reject application with reason("Insufficient income")

// Notificações

notify customer "Your application is being processed"

notify admin "Manual review required"

// Roteamento

route\_to queue\_name

route\_to analyst\_queue with priority("high")

// Aguardar

wait\_for user\_response

wait\_for external\_callback

**Seção @error\_handling**

Define como tratar diferentes tipos de erros:

@error\_handling {

// Erro específico de serviço

CreditBureau.timeout => use\_backup\_credit\_service,

// Erro de categoria

network\_error => retry\_with\_backoff,

// Erro genérico por serviço

PaymentService.\* => escalate\_to\_finance\_team,

// Handler global

\* => log\_error\_and\_continue,

// Múltiplas ações

critical\_system\_error => {

notify admin "Critical error occurred",

escalate\_to\_engineering,

pause\_workflow

}

}

**Seção @config**

Configurações do workflow:

@config {

// Timeouts

timeout: 30m,

step\_timeout: 5m,

// Retry policies

max\_retries: 3,

retry\_policy: exponential\_backoff,

// Prioridades

default\_priority: medium,

escalation\_timeout: 24h,

// IA e otimização

ai\_optimization: true,

learning\_mode: active,

// Monitoramento

audit\_all\_steps: true,

metrics\_enabled: true,

// Concorrência

max\_concurrent\_instances: 100,

batch\_size: 10

}

**Compilação e Execução**

**Processo de Compilação**

O FlowScript passa por várias etapas de compilação:

Código FlowScript (.fs)

↓

[Lexer] → Tokens

↓

[Parser] → AST (Abstract Syntax Tree)

↓

[Semantic Analyzer] → Validação e otimização

↓

[Code Generator] → Bytecode executável

↓

[Runtime] → Execução

**Compilador FlowScript**

from flowscript import FlowScriptCompiler

# Inicializar compilador

compiler = FlowScriptCompiler()

# Compilar de arquivo

workflow = compiler.compile\_file("workflow.fs")

# Compilar de string

code = """

@workflow Example {

@flow {

start {

notify "Hello World"

}

}

}

"""

workflow = compiler.compile(code)

# Compilar com opções

workflow = compiler.compile(code, options={

"optimize": True,

"debug": False,

"strict\_mode": True

})

**Opções de Compilação**

compile\_options = {

"optimize": True, # Habilita otimizações

"debug": False, # Modo debug

"strict\_mode": True, # Validação rigorosa

"target\_version": "1.0", # Versão alvo

"warnings\_as\_errors": False, # Trata warnings como erros

"generate\_docs": True, # Gera documentação automática

"validate\_plugins": True # Valida plugins referenciados

}

**Runtime de Execução**

from flowscript import FlowScriptRuntime

# Inicializar runtime

runtime = FlowScriptRuntime()

# Carregar workflow compilado

runtime.load\_workflow(workflow)

# Registrar serviços

runtime.register\_service("CreditBureau", credit\_bureau\_service)

runtime.register\_service("EmailService", email\_service)

# Executar workflow

instance\_id = await runtime.start\_workflow(

workflow\_name="LoanApproval",

context={

"customer": customer\_data,

"application": loan\_application

}

)

# Monitorar execução

status = runtime.get\_workflow\_status(instance\_id)

print(f"Status: {status}")

**Configuração do Runtime**

runtime\_config = {

"max\_concurrent\_workflows": 1000,

"event\_queue\_size": 10000,

"step\_timeout\_default": 300, # 5 minutos

"retry\_attempts": 3,

"metrics\_enabled": True,

"audit\_enabled": True,

"plugin\_sandbox": True

}

runtime = FlowScriptRuntime(config=runtime\_config)

**Execução de Workflows**

**Execução Síncrona**

# Para workflows simples que completam rapidamente

result = runtime.execute\_workflow\_sync("WorkflowName", context)

**Execução Assíncrona**

# Para workflows longos ou com esperas

instance\_id = await runtime.start\_workflow("WorkflowName", context)

# Aguardar conclusão

result = await runtime.wait\_for\_completion(instance\_id)

**Execução em Lote**

# Para processar múltiplos workflows

batch\_results = await runtime.execute\_batch([

{"workflow": "Approval", "context": context1},

{"workflow": "Approval", "context": context2},

{"workflow": "Approval", "context": context3}

])

**Sistema de Plugins**

**Arquitetura de Plugins**

Os plugins no FlowScript são componentes modulares que estendem a funcionalidade do sistema. Cada plugin implementa uma interface padronizada e pode ser carregado dinamicamente.

**Interface de Plugin**

from abc import ABC, abstractmethod

from typing import Any, Dict, List, Optional

class FlowScriptPlugin(ABC):

"""Interface base para plugins FlowScript"""

@property

@abstractmethod

def name(self) -> str:

"""Nome único do plugin"""

pass

@property

@abstractmethod

def version(self) -> str:

"""Versão do plugin"""

pass

@property

@abstractmethod

def description(self) -> str:

"""Descrição do plugin"""

pass

@property

def dependencies(self) -> List[str]:

"""Lista de dependências (outros plugins ou serviços)"""

return []

@property

def parameters(self) -> List[Dict[str, Any]]:

"""Schema dos parâmetros aceitos"""

return []

@abstractmethod

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa o plugin"""

pass

@abstractmethod

async def execute(self, \*args, \*\*kwargs) -> Any:

"""Executa a funcionalidade principal do plugin"""

pass

async def cleanup(self) -> None:

"""Limpeza de recursos (opcional)"""

pass

def validate\_parameters(self, \*args, \*\*kwargs) -> bool:

"""Valida parâmetros de entrada (opcional)"""

return True

**Criando um Plugin Simples**

from flowscript.plugins import FlowScriptPlugin

import aiohttp

import asyncio

class EmailSenderPlugin(FlowScriptPlugin):

"""Plugin para envio de emails"""

@property

def name(self) -> str:

return "EmailSender"

@property

def version(self) -> str:

return "1.0.0"

@property

def description(self) -> str:

return "Envia emails através de serviços SMTP ou APIs"

@property

def parameters(self) -> List[Dict[str, Any]]:

return [

{

"name": "recipient",

"type": "string",

"required": True,

"description": "Email do destinatário"

},

{

"name": "subject",

"type": "string",

"required": True,

"description": "Assunto do email"

},

{

"name": "body",

"type": "string",

"required": True,

"description": "Corpo do email"

},

{

"name": "template",

"type": "string",

"required": False,

"description": "Template a ser usado"

}

]

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa configurações do email"""

self.smtp\_server = config.get("smtp\_server", "localhost")

self.smtp\_port = config.get("smtp\_port", 587)

self.username = config.get("username")

self.password = config.get("password")

self.api\_key = config.get("api\_key")

self.service\_type = config.get("service\_type", "smtp") # smtp, sendgrid, ses

print(f"EmailSender plugin inicializado ({self.service\_type})")

async def execute(self, recipient: str, subject: str, body: str,

template: Optional[str] = None) -> Dict[str, Any]:

"""Envia email"""

# Validar entrada

if not self.validate\_parameters(recipient, subject, body):

raise ValueError("Parâmetros inválidos para envio de email")

# Aplicar template se especificado

if template:

body = await self.\_apply\_template(template, body)

# Enviar baseado no tipo de serviço

if self.service\_type == "smtp":

return await self.\_send\_via\_smtp(recipient, subject, body)

elif self.service\_type == "sendgrid":

return await self.\_send\_via\_sendgrid(recipient, subject, body)

elif self.service\_type == "ses":

return await self.\_send\_via\_ses(recipient, subject, body)

else:

raise ValueError(f"Tipo de serviço não suportado: {self.service\_type}")

def validate\_parameters(self, recipient: str, subject: str, body: str) -> bool:

"""Valida parâmetros de email"""

import re

# Validar formato do email

email\_pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

if not re.match(email\_pattern, recipient):

return False

# Validar campos obrigatórios

if not subject.strip() or not body.strip():

return False

return True

async def \_apply\_template(self, template: str, data: str) -> str:

"""Aplica template ao corpo do email"""

# Implementação simplificada - em produção usar Jinja2 ou similar

templates = {

"welcome": "Bem-vindo! {data}",

"approval": "Parabéns! Sua solicitação foi aprovada. {data}",

"rejection": "Lamentamos informar que sua solicitação foi rejeitada. {data}"

}

template\_body = templates.get(template, "{data}")

return template\_body.format(data=data)

async def \_send\_via\_smtp(self, recipient: str, subject: str, body: str) -> Dict[str, Any]:

"""Envia email via SMTP"""

# Simula envio SMTP

await asyncio.sleep(0.1) # Simula latência

print(f"📧 SMTP: Enviando email para {recipient}")

return {

"status": "sent",

"message\_id": f"smtp\_{int(asyncio.get\_event\_loop().time())}",

"service": "smtp"

}

async def \_send\_via\_sendgrid(self, recipient: str, subject: str, body: str) -> Dict[str, Any]:

"""Envia email via SendGrid API"""

# Implementação real usaria SendGrid API

await asyncio.sleep(0.2) # Simula latência da API

print(f"📧 SendGrid: Enviando email para {recipient}")

return {

"status": "sent",

"message\_id": f"sg\_{int(asyncio.get\_event\_loop().time())}",

"service": "sendgrid"

}

async def \_send\_via\_ses(self, recipient: str, subject: str, body: str) -> Dict[str, Any]:

"""Envia email via AWS SES"""

# Implementação real usaria boto3

await asyncio.sleep(0.15) # Simula latência da AWS

print(f"📧 AWS SES: Enviando email para {recipient}")

return {

"status": "sent",

"message\_id": f"ses\_{int(asyncio.get\_event\_loop().time())}",

"service": "aws\_ses"

}

**Plugin de Integração com API**

class APIConnectorPlugin(FlowScriptPlugin):

"""Plugin genérico para chamadas de API"""

@property

def name(self) -> str:

return "APIConnector"

@property

def version(self) -> str:

return "1.0.0"

@property

def description(self) -> str:

return "Conecta com APIs REST de forma genérica"

@property

def parameters(self) -> List[Dict[str, Any]]:

return [

{"name": "url", "type": "string", "required": True},

{"name": "method", "type": "string", "required": False, "default": "GET"},

{"name": "headers", "type": "object", "required": False},

{"name": "payload", "type": "object", "required": False},

{"name": "timeout", "type": "number", "required": False, "default": 30}

]

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa cliente HTTP"""

self.session = aiohttp.ClientSession(

timeout=aiohttp.ClientTimeout(total=config.get("default\_timeout", 30))

)

self.default\_headers = config.get("default\_headers", {})

async def execute(self, url: str, method: str = "GET",

headers: Optional[Dict] = None,

payload: Optional[Dict] = None,

timeout: int = 30) -> Dict[str, Any]:

"""Executa chamada de API"""

# Monta headers

request\_headers = {\*\*self.default\_headers}

if headers:

request\_headers.update(headers)

try:

async with self.session.request(

method=method.upper(),

url=url,

headers=request\_headers,

json=payload,

timeout=aiohttp.ClientTimeout(total=timeout)

) as response:

result = {

"status\_code": response.status,

"headers": dict(response.headers),

"success": 200 <= response.status < 300

}

# Tenta parsear JSON, senão retorna texto

try:

result["data"] = await response.json()

except:

result["data"] = await response.text()

return result

except asyncio.TimeoutError:

raise Exception(f"Timeout na chamada para {url}")

except Exception as e:

raise Exception(f"Erro na chamada de API: {str(e)}")

async def cleanup(self) -> None:

"""Fecha sessão HTTP"""

if self.session:

await self.session.close()

**Plugin de Processamento de Dados**

class DataProcessorPlugin(FlowScriptPlugin):

"""Plugin para processamento de dados"""

@property

def name(self) -> str:

return "DataProcessor"

@property

def version(self) -> str:

return "1.0.0"

@property

def description(self) -> str:

return "Processa e transforma dados"

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa processador"""

self.transformations = config.get("transformations", {})

self.validators = config.get("validators", {})

async def execute(self, data: Any, operation: str, \*\*kwargs) -> Any:

"""Executa operação nos dados"""

if operation == "validate":

return await self.\_validate\_data(data, kwargs.get("schema"))

elif operation == "transform":

return await self.\_transform\_data(data, kwargs.get("transformation"))

elif operation == "aggregate":

return await self.\_aggregate\_data(data, kwargs.get("aggregation"))

elif operation == "filter":

return await self.\_filter\_data(data, kwargs.get("criteria"))

else:

raise ValueError(f"Operação não suportada: {operation}")

async def \_validate\_data(self, data: Any, schema: Dict) -> Dict[str, Any]:

"""Valida dados contra schema"""

errors = []

# Implementação simplificada de validação

if isinstance(data, dict) and isinstance(schema, dict):

for field, field\_schema in schema.items():

if field\_schema.get("required", False) and field not in data:

errors.append(f"Campo obrigatório '{field}' ausente")

if field in data:

expected\_type = field\_schema.get("type")

if expected\_type == "string" and not isinstance(data[field], str):

errors.append(f"Campo '{field}' deve ser string")

elif expected\_type == "number" and not isinstance(data[field], (int, float)):

errors.append(f"Campo '{field}' deve ser número")

return {

"valid": len(errors) == 0,

"errors": errors,

"data": data

}

async def \_transform\_data(self, data: Any, transformation: str) -> Any:

"""Transforma dados"""

transformations = {

"uppercase": lambda x: x.upper() if isinstance(x, str) else x,

"lowercase": lambda x: x.lower() if isinstance(x, str) else x,

"strip": lambda x: x.strip() if isinstance(x, str) else x,

"round": lambda x: round(x, 2) if isinstance(x, (int, float)) else x

}

transform\_func = transformations.get(transformation)

if not transform\_func:

raise ValueError(f"Transformação não encontrada: {transformation}")

if isinstance(data, list):

return [transform\_func(item) for item in data]

elif isinstance(data, dict):

return {k: transform\_func(v) for k, v in data.items()}

else:

return transform\_func(data)

async def \_aggregate\_data(self, data: List, aggregation: str) -> Any:

"""Agrega dados"""

if not isinstance(data, list):

raise ValueError("Agregação requer lista de dados")

if aggregation == "sum":

return sum(x for x in data if isinstance(x, (int, float)))

elif aggregation == "avg":

numbers = [x for x in data if isinstance(x, (int, float))]

return sum(numbers) / len(numbers) if numbers else 0

elif aggregation == "count":

return len(data)

elif aggregation == "max":

numbers = [x for x in data if isinstance(x, (int, float))]

return max(numbers) if numbers else None

elif aggregation == "min":

numbers = [x for x in data if isinstance(x, (int, float))]

return min(numbers) if numbers else None

else:

raise ValueError(f"Agregação não suportada: {aggregation}")

async def \_filter\_data(self, data: List, criteria: Dict) -> List:

"""Filtra dados baseado em critérios"""

if not isinstance(data, list):

raise ValueError("Filtro requer lista de dados")

filtered = []

for item in data:

if self.\_matches\_criteria(item, criteria):

filtered.append(item)

return filtered

def \_matches\_criteria(self, item: Any, criteria: Dict) -> bool:

"""Verifica se item atende aos critérios"""

if not isinstance(item, dict) or not isinstance(criteria, dict):

return True

for field, condition in criteria.items():

if field not in item:

return False

value = item[field]

if isinstance(condition, dict):

# Condições complexas como {"gt": 100, "lt": 200}

for op, expected in condition.items():

if op == "gt" and not (value > expected):

return False

elif op == "lt" and not (value < expected):

return False

elif op == "eq" and not (value == expected):

return False

elif op == "ne" and not (value != expected):

return False

else:

# Condição simples de igualdade

if value != condition:

return False

return True

**Registrando Plugins**

from flowscript import FlowScriptRuntime

# Inicializar runtime

runtime = FlowScriptRuntime()

# Registrar plugin individual

email\_plugin = EmailSenderPlugin()

await email\_plugin.initialize({

"service\_type": "sendgrid",

"api\_key": "your\_api\_key"

})

runtime.register\_plugin(email\_plugin)

# Registrar múltiplos plugins

plugins = [

EmailSenderPlugin(),

APIConnectorPlugin(),

DataProcessorPlugin()

]

for plugin in plugins:

await plugin.initialize({})

runtime.register\_plugin(plugin)

# Registrar plugin com configuração

runtime.register\_plugin\_with\_config("DatabaseConnector", {

"connection\_string": "postgresql://user:pass@localhost/db",

"pool\_size": 10,

"timeout": 30

})

# Carregar plugins de diretório

runtime.load\_plugins\_from\_directory("./plugins/")

# Listar plugins registrados

available\_plugins = runtime.list\_plugins()

print("Plugins disponíveis:", available\_plugins)

**Plugin Loader e Discovery**

import os

import importlib.util

from typing import List, Type

class PluginLoader:

"""Carregador automático de plugins"""

def \_\_init\_\_(self, plugin\_directories: List[str]):

self.plugin\_directories = plugin\_directories

self.loaded\_plugins: Dict[str, FlowScriptPlugin] = {}

async def discover\_plugins(self) -> List[Type[FlowScriptPlugin]]:

"""Descobre plugins nos diretórios especificados"""

plugin\_classes = []

for directory in self.plugin\_directories:

if not os.path.exists(directory):

continue

for filename in os.listdir(directory):

if filename.endswith('.py') and not filename.startswith('\_\_'):

plugin\_path = os.path.join(directory, filename)

plugin\_classes.extend(self.\_load\_plugins\_from\_file(plugin\_path))

return plugin\_classes

def \_load\_plugins\_from\_file(self, file\_path: str) -> List[Type[FlowScriptPlugin]]:

"""Carrega plugins de um arquivo Python"""

spec = importlib.util.spec\_from\_file\_location("plugin\_module", file\_path)

module = importlib.util.module\_from\_spec(spec)

spec.loader.exec\_module(module)

plugin\_classes = []

for attr\_name in dir(module):

attr = getattr(module, attr\_name)

if (isinstance(attr, type) and

issubclass(attr, FlowScriptPlugin) and

attr != FlowScriptPlugin):

plugin\_classes.append(attr)

return plugin\_classes

async def load\_all\_plugins(self, runtime: FlowScriptRuntime,

config: Dict[str, Dict] = None) -> None:

"""Carrega todos os plugins descobertos"""

plugin\_classes = await self.discover\_plugins()

for plugin\_class in plugin\_classes:

try:

plugin = plugin\_class()

plugin\_config = config.get(plugin.name, {}) if config else {}

await plugin.initialize(plugin\_config)

runtime.register\_plugin(plugin)

self.loaded\_plugins[plugin.name] = plugin

print(f"✅ Plugin '{plugin.name}' carregado com sucesso")

except Exception as e:

print(f"❌ Erro ao carregar plugin '{plugin\_class.\_\_name\_\_}': {e}")

# Exemplo de uso

loader = PluginLoader(["./plugins", "./custom\_plugins"])

await loader.load\_all\_plugins(runtime, {

"EmailSender": {"service\_type": "sendgrid", "api\_key": "key"},

"DatabaseConnector": {"connection\_string": "sqlite:///app.db"}

})

**Integração com Serviços**

**Sistema de Serviços**

O FlowScript distingue entre **plugins** (componentes internos) e **serviços** (sistemas externos). Serviços são registrados separadamente e podem representar APIs, bancos de dados, sistemas de mensageria, etc.

**Criando um Serviço**

from typing import Any, Dict, List, Optional

import asyncio

import aiohttp

class CreditBureauService:

"""Serviço para integração com bureau de crédito"""

def \_\_init\_\_(self, api\_url: str, api\_key: str, timeout: int = 30):

self.api\_url = api\_url

self.api\_key = api\_key

self.timeout = timeout

self.session = None

async def initialize(self):

"""Inicializa conexão com serviço"""

self.session = aiohttp.ClientSession(

timeout=aiohttp.ClientTimeout(total=self.timeout),

headers={"Authorization": f"Bearer {self.api\_key}"}

)

print("CreditBureau service initialized")

async def getScore(self, ssn: str) -> int:

"""Obtém score de crédito"""

url = f"{self.api\_url}/credit-score"

payload = {"ssn": ssn}

try:

async with self.session.post(url, json=payload) as response:

if response.status == 200:

data = await response.json()

return data.get("score", 0)

else:

raise Exception(f"API returned status {response.status}")

except asyncio.TimeoutError:

raise Exception("Credit bureau timeout")

except Exception as e:

raise Exception(f"Credit bureau error: {str(e)}")

async def getFullReport(self, ssn: str) -> Dict[str, Any]:

"""Obtém relatório completo de crédito"""

url = f"{self.api\_url}/full-report"

payload = {"ssn": ssn}

async with self.session.post(url, json=payload) as response:

return await response.json()

async def close(self):

"""Fecha conexões"""

if self.session:

await self.session.close()

class DatabaseService:

"""Serviço genérico para banco de dados"""

def \_\_init\_\_(self, connection\_string: str):

self.connection\_string = connection\_string

self.pool = None

async def initialize(self):

"""Inicializa pool de conexões"""

# Em implementação real, usaria SQLAlchemy ou asyncpg

print(f"Database service initialized: {self.connection\_string}")

self.pool = "mock\_pool" # Mock para exemplo

async def execute\_query(self, query: str, parameters: List = None) -> List[Dict]:

"""Executa query SELECT"""

print(f"Executing query: {query}")

if parameters:

print(f"Parameters: {parameters}")

# Mock result - em produção, executaria query real

return [{"id": 1, "result": "mock\_data"}]

async def execute\_command(self, command: str, parameters: List = None) -> int:

"""Executa comando INSERT/UPDATE/DELETE"""

print(f"Executing command: {command}")

if parameters:

print(f"Parameters: {parameters}")

# Mock - retorna número de linhas afetadas

return 1

async def close(self):

"""Fecha pool de conexões"""

print("Database connections closed")

class MessageQueueService:

"""Serviço para sistema de mensageria"""

def \_\_init\_\_(self, broker\_url: str):

self.broker\_url = broker\_url

self.connection = None

async def initialize(self):

"""Conecta ao broker"""

print(f"MessageQueue service initialized: {self.broker\_url}")

self.connection = "mock\_connection"

async def publish(self, queue: str, message: Dict[str, Any]) -> bool:

"""Publica mensagem em fila"""

print(f"Publishing to {queue}: {message}")

await asyncio.sleep(0.1) # Simula latência

return True

async def subscribe(self, queue: str, callback: callable) -> None:

"""Inscreve-se em fila para receber mensagens"""

print(f"Subscribing to queue: {queue}")

# Em implementação real, configuraria callback

async def close(self):

"""Fecha conexão"""

print("MessageQueue connection closed")

**Registrando Serviços**

from flowscript import FlowScriptRuntime

async def setup\_services(runtime: FlowScriptRuntime):

"""Configura todos os serviços necessários"""

# Criar e inicializar serviços

credit\_bureau = CreditBureauService(

api\_url="https://api.creditbureau.com",

api\_key="your\_api\_key"

)

await credit\_bureau.initialize()

database = DatabaseService("postgresql://user:pass@localhost/flowscript")

await database.initialize()

message\_queue = MessageQueueService("redis://localhost:6379")

await message\_queue.initialize()

# Registrar no runtime

runtime.register\_service("CreditBureau", credit\_bureau)

runtime.register\_service("Database", database)

runtime.register\_service("MessageQueue", message\_queue)

print("✅ Todos os serviços registrados")

# Usar em workflow

runtime = FlowScriptRuntime()

await setup\_services(runtime)

**Service Discovery e Health Checks**

class ServiceRegistry:

"""Registry para descoberta e monitoramento de serviços"""

def \_\_init\_\_(self):

self.services: Dict[str, Any] = {}

self.health\_status: Dict[str, bool] = {}

self.last\_health\_check: Dict[str, float] = {}

def register\_service(self, name: str, service: Any, health\_check\_interval: int = 60):

"""Registra serviço com health check"""

self.services[name] = service

self.health\_status[name] = True

self.last\_health\_check[name] = 0

# Agendar health checks periódicos

asyncio.create\_task(self.\_periodic\_health\_check(name, health\_check\_interval))

async def \_periodic\_health\_check(self, service\_name: str, interval: int):

"""Executa health check periódico"""

while service\_name in self.services:

try:

service = self.services[service\_name]

if hasattr(service, 'health\_check'):

is\_healthy = await service.health\_check()

else:

is\_healthy = True # Assume saudável se não tem health check

self.health\_status[service\_name] = is\_healthy

self.last\_health\_check[service\_name] = asyncio.get\_event\_loop().time()

if not is\_healthy:

print(f"⚠️ Service {service\_name} is unhealthy")

except Exception as e:

print(f"❌ Health check failed for {service\_name}: {e}")

self.health\_status[service\_name] = False

await asyncio.sleep(interval)

def get\_healthy\_services(self) -> List[str]:

"""Retorna lista de serviços saudáveis"""

return [name for name, healthy in self.health\_status.items() if healthy]

def get\_service\_status(self) -> Dict[str, Dict]:

"""Retorna status completo de todos os serviços"""

status = {}

for name in self.services:

status[name] = {

"healthy": self.health\_status.get(name, False),

"last\_check": self.last\_health\_check.get(name, 0)

}

return status

# Integração com runtime

class EnhancedFlowScriptRuntime(FlowScriptRuntime):

"""Runtime com service discovery integrado"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.service\_registry = ServiceRegistry()

def register\_service(self, name: str, service: Any, health\_check\_interval: int = 60):

"""Registra serviço com health monitoring"""

super().register\_service(name, service)

self.service\_registry.register\_service(name, service, health\_check\_interval)

async def get\_service\_health(self) -> Dict[str, Dict]:

"""Retorna status de saúde dos serviços"""

return self.service\_registry.get\_service\_status()

**Tratamento de Erros**

**Estratégias de Error Handling**

O FlowScript oferece múltiplas estratégias para tratamento de erros, desde recuperação automática até escalação para intervenção humana.

**Tipos de Erros**

@error\_handling {

// 1. Erros específicos de serviço

CreditBureau.timeout => retry\_with\_backoff,

CreditBureau.rate\_limit => wait\_and\_retry,

CreditBureau.unauthorized => refresh\_credentials,

// 2. Erros por categoria

network\_error => switch\_to\_backup\_service,

validation\_error => request\_data\_correction,

authorization\_error => escalate\_to\_admin,

// 3. Erros por padrão

\*.timeout => retry\_operation,

\*.connection\_error => check\_service\_health,

// 4. Tratamento em cascata

critical\_system\_error => {

log\_critical\_error,

notify\_engineering\_team,

pause\_workflow,

escalate\_to\_on\_call

},

// 5. Fallback global

\* => log\_and\_continue\_with\_degraded\_service

}

**Políticas de Retry**

from typing import Callable, Any

import asyncio

import random

class RetryPolicy:

"""Políticas de retry para operações"""

@staticmethod

async def exponential\_backoff(func: Callable, max\_attempts: int = 3,

base\_delay: float = 1.0, max\_delay: float = 60.0):

"""Retry com backoff exponencial"""

for attempt in range(max\_attempts):

try:

return await func()

except Exception as e:

if attempt == max\_attempts - 1:

raise e

# Calcular delay com jitter

delay = min(base\_delay \* (2 \*\* attempt), max\_delay)

jitter = random.uniform(0, 0.1) \* delay

total\_delay = delay + jitter

print(f"Attempt {attempt + 1} failed, retrying in {total\_delay:.2f}s")

await asyncio.sleep(total\_delay)

@staticmethod

async def fixed\_interval(func: Callable, max\_attempts: int = 3, interval: float = 5.0):

"""Retry com intervalo fixo"""

for attempt in range(max\_attempts):

try:

return await func()

except Exception as e:

if attempt == max\_attempts - 1:

raise e

print(f"Attempt {attempt + 1} failed, retrying in {interval}s")

await asyncio.sleep(interval)

@staticmethod

async def linear\_backoff(func: Callable, max\_attempts: int = 3, base\_delay: float = 1.0):

"""Retry com backoff linear"""

for attempt in range(max\_attempts):

try:

return await func()

except Exception as e:

if attempt == max\_attempts - 1:

raise e

delay = base\_delay \* (attempt + 1)

print(f"Attempt {attempt + 1} failed, retrying in {delay}s")

await asyncio.sleep(delay)

class CircuitBreaker:

"""Implementa padrão Circuit Breaker"""

def \_\_init\_\_(self, failure\_threshold: int = 5, recovery\_timeout: int = 60):

self.failure\_threshold = failure\_threshold

self.recovery\_timeout = recovery\_timeout

self.failure\_count = 0

self.last\_failure\_time = 0

self.state = "CLOSED" # CLOSED, OPEN, HALF\_OPEN

async def call(self, func: Callable, \*args, \*\*kwargs):

"""Executa função com circuit breaker"""

if self.state == "OPEN":

if asyncio.get\_event\_loop().time() - self.last\_failure\_time > self.recovery\_timeout:

self.state = "HALF\_OPEN"

print("Circuit breaker transitioning to HALF\_OPEN")

else:

raise Exception("Circuit breaker is OPEN")

try:

result = await func(\*args, \*\*kwargs)

if self.state == "HALF\_OPEN":

self.state = "CLOSED"

self.failure\_count = 0

print("Circuit breaker CLOSED - service recovered")

return result

except Exception as e:

self.failure\_count += 1

self.last\_failure\_time = asyncio.get\_event\_loop().time()

if self.failure\_count >= self.failure\_threshold:

self.state = "OPEN"

print(f"Circuit breaker OPEN - {self.failure\_count} failures")

raise e

class ErrorRecoveryManager:

"""Gerenciador de recuperação de erros"""

def \_\_init\_\_(self):

self.circuit\_breakers: Dict[str, CircuitBreaker] = {}

self.retry\_policies: Dict[str, Callable] = {

"exponential\_backoff": RetryPolicy.exponential\_backoff,

"fixed\_interval": RetryPolicy.fixed\_interval,

"linear\_backoff": RetryPolicy.linear\_backoff

}

def get\_circuit\_breaker(self, service\_name: str) -> CircuitBreaker:

"""Obtém circuit breaker para serviço"""

if service\_name not in self.circuit\_breakers:

self.circuit\_breakers[service\_name] = CircuitBreaker()

return self.circuit\_breakers[service\_name]

async def execute\_with\_recovery(self, func: Callable, service\_name: str,

retry\_policy: str = "exponential\_backoff",

use\_circuit\_breaker: bool = True) -> Any:

"""Executa função com estratégias de recuperação"""

if use\_circuit\_breaker:

circuit\_breaker = self.get\_circuit\_breaker(service\_name)

async def protected\_func():

return await circuit\_breaker.call(func)

else:

protected\_func = func

retry\_func = self.retry\_policies.get(retry\_policy, RetryPolicy.exponential\_backoff)

return await retry\_func(protected\_func)

**Implementação no Runtime**

class FlowScriptRuntimeWithErrorHandling(FlowScriptRuntime):

"""Runtime com tratamento avançado de erros"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.error\_recovery = ErrorRecoveryManager()

self.error\_handlers: Dict[str, Callable] = {}

self.global\_error\_handler: Optional[Callable] = None

def register\_error\_handler(self, error\_pattern: str, handler: Callable):

"""Registra handler específico para tipo de erro"""

self.error\_handlers[error\_pattern] = handler

def set\_global\_error\_handler(self, handler: Callable):

"""Define handler global para erros não tratados"""

self.global\_error\_handler = handler

async def handle\_error(self, error: Exception, context: Dict[str, Any]) -> Any:

"""Processa erro baseado nas regras definidas"""

error\_type = type(error).\_\_name\_\_

service\_name = context.get("service\_name", "unknown")

# Procura handler específico

for pattern, handler in self.error\_handlers.items():

if self.\_matches\_error\_pattern(error, pattern):

print(f"Handling error with pattern: {pattern}")

return await handler(error, context)

# Handler global

if self.global\_error\_handler:

return await self.global\_error\_handler(error, context)

# Fallback padrão

print(f"Unhandled error: {error}")

raise error

def \_matches\_error\_pattern(self, error: Exception, pattern: str) -> bool:

"""Verifica se erro corresponde ao padrão"""

error\_type = type(error).\_\_name\_\_

error\_message = str(error)

# Padrões específicos

if pattern == "\*":

return True

elif "." in pattern:

service, error\_name = pattern.split(".", 1)

return error\_name in error\_message or error\_name == error\_type

else:

return pattern in error\_message or pattern == error\_type

async def execute\_with\_error\_handling(self, func: Callable, context: Dict[str, Any]) -> Any:

"""Executa função com tratamento de erro completo"""

try:

return await func()

except Exception as e:

return await self.handle\_error(e, context)

# Exemplo de handlers de erro

async def handle\_timeout\_error(error: Exception, context: Dict[str, Any]) -> Any:

"""Handler para erros de timeout"""

service\_name = context.get("service\_name", "unknown")

print(f"Handling timeout for service: {service\_name}")

# Implementar estratégia específica para timeout

if service\_name == "CreditBureau":

# Usar serviço backup

backup\_service = context.get("backup\_service")

if backup\_service:

return await backup\_service.getScore(context.get("ssn"))

# Fallback genérico

raise Exception("Service temporarily unavailable")

async def handle\_rate\_limit\_error(error: Exception, context: Dict[str, Any]) -> Any:

"""Handler para rate limiting"""

print("Rate limit detected, implementing backoff strategy")

# Aguardar antes de retry

await asyncio.sleep(60) # 1 minuto

# Retry da operação original

original\_func = context.get("original\_function")

if original\_func:

return await original\_func()

raise error

# Configuração de error handling

runtime = FlowScriptRuntimeWithErrorHandling()

runtime.register\_error\_handler("timeout", handle\_timeout\_error)

runtime.register\_error\_handler("rate\_limit", handle\_rate\_limit\_error)

**Monitoramento e Debugging**

**Sistema de Métricas**

from dataclasses import dataclass

from typing import Dict, List, Any

import time

import json

@dataclass

class Metric:

"""Representação de uma métrica"""

name: str

value: Any

timestamp: float

tags: Dict[str, str]

class MetricsCollector:

"""Coletor de métricas para workflows"""

def \_\_init\_\_(self):

self.metrics: List[Metric] = []

self.counters: Dict[str, int] = {}

self.gauges: Dict[str, float] = {}

self.histograms: Dict[str, List[float]] = {}

def increment\_counter(self, name: str, tags: Dict[str, str] = None):

"""Incrementa contador"""

self.counters[name] = self.counters.get(name, 0) + 1

self.\_record\_metric(name, self.counters[name], tags or {})

def set\_gauge(self, name: str, value: float, tags: Dict[str, str] = None):

"""Define valor de gauge"""

self.gauges[name] = value

self.\_record\_metric(name, value, tags or {})

def record\_histogram(self, name: str, value: float, tags: Dict[str, str] = None):

"""Registra valor em histograma"""

if name not in self.histograms:

self.histograms[name] = []

self.histograms[name].append(value)

self.\_record\_metric(name, value, tags or {})

def \_record\_metric(self, name: str, value: Any, tags: Dict[str, str]):

"""Registra métrica com timestamp"""

metric = Metric(

name=name,

value=value,

timestamp=time.time(),

tags=tags

)

self.metrics.append(metric)

def get\_summary(self) -> Dict[str, Any]:

"""Retorna resumo das métricas"""

summary = {

"counters": self.counters.copy(),

"gauges": self.gauges.copy(),

"histograms": {}

}

# Calcular estatísticas dos histogramas

for name, values in self.histograms.items():

if values:

summary["histograms"][name] = {

"count": len(values),

"min": min(values),

"max": max(values),

"avg": sum(values) / len(values),

"p50": self.\_percentile(values, 50),

"p95": self.\_percentile(values, 95),

"p99": self.\_percentile(values, 99)

}

return summary

def \_percentile(self, values: List[float], percentile: int) -> float:

"""Calcula percentil"""

sorted\_values = sorted(values)

index = int(len(sorted\_values) \* percentile / 100)

return sorted\_values[min(index, len(sorted\_values) - 1)]

class WorkflowTracer:

"""Sistema de tracing para workflows"""

def \_\_init\_\_(self):

self.traces: Dict[str, Dict] = {}

self.active\_spans: Dict[str, str] = {}

def start\_trace(self, trace\_id: str, workflow\_name: str) -> None:

"""Inicia novo trace"""

self.traces[trace\_id] = {

"workflow\_name": workflow\_name,

"start\_time": time.time(),

"spans": [],

"status": "running"

}

def start\_span(self, trace\_id: str, span\_name: str, parent\_span: str = None) -> str:

"""Inicia novo span dentro de um trace"""

span\_id = f"{span\_name}\_{int(time.time() \* 1000)}"

span = {

"span\_id": span\_id,

"name": span\_name,

"start\_time": time.time(),

"parent\_span": parent\_span,

"tags": {},

"logs": []

}

if trace\_id in self.traces:

self.traces[trace\_id]["spans"].append(span)

self.active\_spans[span\_id] = trace\_id

return span\_id

def finish\_span(self, span\_id: str, status: str = "success") -> None:

"""Finaliza span"""

if span\_id in self.active\_spans:

trace\_id = self.active\_spans[span\_id]

trace = self.traces[trace\_id]

for span in trace["spans"]:

if span["span\_id"] == span\_id:

span["end\_time"] = time.time()

span["duration"] = span["end\_time"] - span["start\_time"]

span["status"] = status

break

del self.active\_spans[span\_id]

def add\_span\_tag(self, span\_id: str, key: str, value: str) -> None:

"""Adiciona tag ao span"""

if span\_id in self.active\_spans:

trace\_id = self.active\_spans[span\_id]

trace = self.traces[trace\_id]

for span in trace["spans"]:

if span["span\_id"] == span\_id:

span["tags"][key] = value

break

def add\_span\_log(self, span\_id: str, message: str, level: str = "info") -> None:

"""Adiciona log ao span"""

if span\_id in self.active\_spans:

trace\_id = self.active\_spans[span\_id]

trace = self.traces[trace\_id]

for span in trace["spans"]:

if span["span\_id"] == span\_id:

span["logs"].append({

"timestamp": time.time(),

"message": message,

"level": level

})

break

def finish\_trace(self, trace\_id: str, status: str = "success") -> None:

"""Finaliza trace"""

if trace\_id in self.traces:

trace = self.traces[trace\_id]

trace["end\_time"] = time.time()

trace["duration"] = trace["end\_time"] - trace["start\_time"]

trace["status"] = status

def get\_trace(self, trace\_id: str) -> Dict:

"""Retorna dados do trace"""

return self.traces.get(trace\_id, {})

class FlowScriptDebugger:

"""Debugger avançado para FlowScript"""

def \_\_init\_\_(self, runtime):

self.runtime = runtime

self.breakpoints: Dict[str, List[str]] = {} # workflow\_name -> [steps]

self.watchpoints: Dict[str, List[str]] = {} # workflow\_name -> [variables]

self.step\_mode = False

self.debug\_sessions: Dict[str, Dict] = {}

def set\_breakpoint(self, workflow\_name: str, step\_name: str) -> None:

"""Define breakpoint em step específico"""

if workflow\_name not in self.breakpoints:

self.breakpoints[workflow\_name] = []

self.breakpoints[workflow\_name].append(step\_name)

print(f"Breakpoint set: {workflow\_name}.{step\_name}")

def set\_watchpoint(self, workflow\_name: str, variable\_name: str) -> None:

"""Define watchpoint em variável"""

if workflow\_name not in self.watchpoints:

self.watchpoints[workflow\_name] = []

self.watchpoints[workflow\_name].append(variable\_name)

print(f"Watchpoint set: {workflow\_name}.{variable\_name}")

def start\_debug\_session(self, instance\_id: str) -> None:

"""Inicia sessão de debug"""

self.debug\_sessions[instance\_id] = {

"started": time.time(),

"step\_history": [],

"variable\_history": [],

"paused": False

}

async def check\_breakpoint(self, instance\_id: str, workflow\_name: str, step\_name: str) -> bool:

"""Verifica se deve pausar em breakpoint"""

should\_pause = False

# Verifica breakpoints

if (workflow\_name in self.breakpoints and

step\_name in self.breakpoints[workflow\_name]):

shoul

**Exemplos Práticos**

**Exemplo 1: Sistema de Aprovação de Empréstimos**

@workflow LoanApprovalSystem {

@context {

customer: Customer,

loan\_application: LoanApplication,

credit\_score: Number,

risk\_assessment: RiskAnalysis,

decision: ApprovalDecision

}

@triggers {

application\_submitted => validate\_application,

credit\_check\_complete => evaluate\_risk,

risk\_analysis\_done => make\_decision,

manual\_review\_complete => finalize\_decision

}

@flow {

validate\_application {

call DataValidator.validate(loan\_application) -> validation\_result

if validation\_result.valid => get\_credit\_score

else => request\_corrections

}

get\_credit\_score {

call CreditBureau.getScore(customer.ssn) -> credit\_score

call AuditLogger("credit\_check", customer.id, "Score retrieved")

emit credit\_check\_complete

}

evaluate\_risk {

call RiskAnalyzer.analyze(customer, loan\_application, credit\_score) -> risk\_assessment

emit risk\_analysis\_done

}

make\_decision {

branch {

credit\_score >= 750 && risk\_assessment.level == "low" => auto\_approve\_premium,

credit\_score >= 650 && risk\_assessment.level != "high" => auto\_approve\_standard,

credit\_score >= 550 => send\_to\_manual\_review,

credit\_score < 550 => auto\_reject

}

}

auto\_approve\_premium {

approve loan\_application with {

rate: 3.5,

amount: loan\_application.requested\_amount,

terms: "premium"

}

call EmailSender(customer.email, "Loan Approved", "Premium terms applied")

emit loan\_approved

}

auto\_approve\_standard {

approve loan\_application with {

rate: 5.5,

amount: calculate\_approved\_amount(loan\_application.requested\_amount, credit\_score),

terms: "standard"

}

call EmailSender(customer.email, "Loan Approved", "Standard terms applied")

emit loan\_approved

}

send\_to\_manual\_review {

route\_to underwriter\_queue with priority(calculate\_priority(risk\_assessment))

call EmailSender(customer.email, "Under Review", "Manual review in progress")

wait\_for manual\_review\_complete

}

auto\_reject {

reject loan\_application with reason("Credit score below minimum threshold")

call EmailSender(customer.email, "Application Declined", "Unfortunately, we cannot approve your loan at this time")

emit loan\_rejected

}

request\_corrections {

call EmailSender(customer.email, "Additional Information Required", validation\_result.message)

wait\_for application\_resubmitted

}

finalize\_decision {

if decision.approved => send\_approval\_notification

else => send\_rejection\_notification

}

send\_approval\_notification {

call EmailSender(customer.email, "Final Approval", decision.details)

call DatabaseConnector("UPDATE loans SET status='approved', decision\_date=NOW() WHERE id=" + loan\_application.id)

emit loan\_finalized

}

send\_rejection\_notification {

call EmailSender(customer.email, "Application Decision", decision.details)

call DatabaseConnector("UPDATE loans SET status='rejected', decision\_date=NOW() WHERE id=" + loan\_application.id)

emit loan\_finalized

}

}

@error\_handling {

CreditBureau.timeout => use\_backup\_credit\_service,

CreditBureau.rate\_limit => wait\_and\_retry\_credit\_check,

RiskAnalyzer.model\_unavailable => escalate\_to\_senior\_underwriter,

EmailSender.delivery\_failed => log\_notification\_failure,

DatabaseConnector.connection\_error => retry\_database\_operation,

\* => log\_error\_and\_escalate

}

@config {

timeout: 24h,

step\_timeout: 5m,

max\_retries: 3,

retry\_policy: exponential\_backoff,

priority\_queue\_enabled: true,

ai\_optimization: true,

audit\_trail: true

}

}

**Exemplo 2: Workflow de Onboarding de Cliente**

@workflow CustomerOnboarding {

@context {

customer: NewCustomer,

documents: DocumentCollection,

verification\_status: VerificationResult,

account: CustomerAccount

}

@triggers {

registration\_complete => collect\_documents,

documents\_uploaded => verify\_identity,

identity\_verified => create\_account,

account\_created => setup\_services

}

@flow {

collect\_documents {

call EmailSender(customer.email, "Welcome!", "Please upload required documents")

call DocumentCollector.requestDocuments(customer, ["id", "proof\_of\_address", "income\_statement"])

wait\_for documents\_uploaded

}

verify\_identity {

call IdentityVerifier.verify(documents) -> verification\_status

branch {

verification\_status.status == "approved" => create\_account,

verification\_status.status == "rejected" => request\_additional\_documents,

verification\_status.status == "manual\_review" => escalate\_to\_compliance

}

}

create\_account {

call AccountService.createAccount(customer) -> account

call CRMService.addCustomer(customer, account)

emit account\_created

}

setup\_services {

// Parallel service setup

parallel {

setup\_banking\_services,

setup\_digital\_access,

setup\_notifications

}

sync\_point all\_services\_ready {

call EmailSender(customer.email, "Account Ready", "Your account is now fully set up")

emit onboarding\_complete

}

}

setup\_banking\_services {

call BankingService.setupChecking(account)

call BankingService.setupSavings(account)

call CardService.issueDebitCard(account)

}

setup\_digital\_access {

call AuthService.createCredentials(customer)

call MobileAppService.enableAccess(customer)

call OnlineBankingService.activate(account)

}

setup\_notifications {

call NotificationService.setupPreferences(customer)

call AlertService.configureDefaults(account)

}

request\_additional\_documents {

call EmailSender(customer.email, "Additional Documents Required", verification\_status.message)

wait\_for documents\_uploaded

}

escalate\_to\_compliance {

route\_to compliance\_team with priority("high")

call EmailSender(customer.email, "Manual Review", "Your application is under compliance review")

wait\_for manual\_verification\_complete

}

}

@error\_handling {

IdentityVerifier.service\_unavailable => queue\_for\_later\_verification,

AccountService.creation\_failed => escalate\_to\_account\_team,

BankingService.setup\_error => retry\_banking\_setup,

\* => log\_error\_and\_notify\_support

}

@config {

timeout: 72h,

document\_retention: 7d,

compliance\_escalation\_timeout: 48h,

parallel\_execution: true

}

}

**Exemplo 3: Pipeline de Processamento de Dados**

@workflow DataProcessingPipeline {

@context {

raw\_data: DataBatch,

processed\_data: ProcessedBatch,

validation\_report: ValidationReport,

output\_location: String

}

@triggers {

data\_received => validate\_data,

validation\_complete => transform\_data,

transformation\_complete => store\_results

}

@flow {

validate\_data {

call DataValidator.validateSchema(raw\_data) -> validation\_report

if validation\_report.valid => transform\_data

else => handle\_invalid\_data

}

transform\_data {

// Parallel processing for large datasets

call DataProcessor.split(raw\_data, batch\_size: 1000) -> data\_batches

parallel\_foreach batch in data\_batches {

call DataProcessor.transform(batch) -> transformed\_batch

call DataValidator.validateTransformation(transformed\_batch) -> batch\_validation

if batch\_validation.valid => continue

else => quarantine\_batch

}

call DataProcessor.merge(transformed\_batches) -> processed\_data

emit transformation\_complete

}

store\_results {

call StorageService.store(processed\_data, output\_location) -> storage\_result

call MetadataService.updateCatalog(processed\_data, storage\_result)

call NotificationService.notifyCompletion(processed\_data.metadata)

emit processing\_complete

}

handle\_invalid\_data {

call DataProcessor.quarantine(raw\_data, validation\_report.errors)

call AlertService.notifyDataQualityTeam(validation\_report)

emit data\_quarantined

}

quarantine\_batch {

call DataProcessor.quarantineBatch(batch, batch\_validation.errors)

// Continue processing other batches

}

}

@error\_handling {

DataProcessor.memory\_error => scale\_up\_processing\_resources,

StorageService.capacity\_full => provision\_additional\_storage,

network\_timeout => retry\_with\_backoff,

\* => alert\_operations\_team

}

@config {

timeout: 4h,

batch\_size: 1000,

max\_parallel\_batches: 10,

retry\_attempts: 3,

auto\_scaling: true

}

}

**API Reference**

**Core Classes**

**FlowScriptCompiler**

class FlowScriptCompiler:

"""Compilador principal do FlowScript"""

def \_\_init\_\_(self, options: Dict[str, Any] = None):

"""Inicializa compilador com opções"""

pass

def compile(self, source\_code: str, \*\*kwargs) -> WorkflowNode:

"""Compila código FlowScript para AST"""

pass

def compile\_file(self, file\_path: str, \*\*kwargs) -> WorkflowNode:

"""Compila arquivo FlowScript"""

pass

def validate\_syntax(self, source\_code: str) -> ValidationResult:

"""Valida sintaxe sem compilar completamente"""

pass

def get\_dependencies(self, workflow: WorkflowNode) -> List[str]:

"""Extrai dependências do workflow"""

pass

**FlowScriptRuntime**

class FlowScriptRuntime:

"""Runtime de execução do FlowScript"""

def \_\_init\_\_(self, config: Dict[str, Any] = None):

"""Inicializa runtime com configuração"""

pass

async def start\_workflow(self, workflow\_name: str,

context: Dict[str, Any] = None) -> str:

"""Inicia execução de workflow"""

pass

async def emit\_event(self, event\_name: str,

data: Dict[str, Any] = None) -> None:

"""Emite evento para workflows"""

pass

def register\_service(self, name: str, service: Any) -> None:

"""Registra serviço externo"""

pass

def register\_plugin(self, plugin: FlowScriptPlugin) -> None:

"""Registra plugin"""

pass

def get\_workflow\_status(self, instance\_id: str) -> Dict[str, Any]:

"""Retorna status de workflow"""

pass

async def stop\_workflow(self, instance\_id: str) -> None:

"""Para execução de workflow"""

pass

**FlowScriptPlugin**

class FlowScriptPlugin(ABC):

"""Interface base para plugins"""

@property

@abstractmethod

def name(self) -> str:

"""Nome único do plugin"""

pass

@property

@abstractmethod

def version(self) -> str:

"""Versão do plugin"""

pass

@abstractmethod

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa plugin"""

pass

@abstractmethod

async def execute(self, \*args, \*\*kwargs) -> Any:

"""Executa funcionalidade do plugin"""

pass

async def cleanup(self) -> None:

"""Limpeza de recursos"""

pass

**Utilitários**

**Template Engine**

class FlowScriptTemplateEngine:

"""Engine para geração de workflows a partir de templates"""

def generate\_workflow(self, template\_name: str, \*\*kwargs) -> str:

"""Gera workflow a partir de template"""

pass

def list\_templates(self) -> Dict[str, List[str]]:

"""Lista templates disponíveis"""

pass

def register\_template(self, name: str, template: str) -> None:

"""Registra novo template"""

pass

**Debugger**

class FlowScriptDebugger:

"""Debugger para workflows"""

def set\_breakpoint(self, workflow\_name: str, step\_name: str) -> None:

"""Define breakpoint"""

pass

def enable\_step\_mode(self) -> None:

"""Habilita modo step-by-step"""

pass

async def debug\_workflow(self, workflow\_name: str,

context: Dict[str, Any]) -> str:

"""Executa workflow em modo debug"""

pass

**Melhores Práticas**

**Design de Workflows**

**1. Estruturação de Steps**

* **Granularidade**: Mantenha steps pequenos e focados em uma responsabilidade
* **Nomenclatura**: Use nomes descritivos que indicam claramente a ação
* **Reutilização**: Projete steps que possam ser reutilizados em diferentes contextos

// ✅ Boa prática - steps granulares

@flow {

validate\_customer\_data {

call CustomerValidator.validate(customer) -> validation\_result

if validation\_result.valid => check\_credit\_score

else => request\_data\_correction

}

check\_credit\_score {

call CreditBureau.getScore(customer.ssn) -> credit\_score

emit credit\_check\_complete

}

}

// ❌ Evitar - step muito amplo

@flow {

process\_customer {

// Muitas responsabilidades em um step

call CustomerValidator.validate(customer) -> validation\_result

call CreditBureau.getScore(customer.ssn) -> credit\_score

call RiskAnalyzer.analyze(customer, credit\_score) -> risk

// ... mais código

}

}

**2. Gestão de Estado**

* **Context Mínimo**: Mantenha apenas dados essenciais no contexto
* **Imutabilidade**: Evite modificar dados de contexto diretamente
* **Limpeza**: Remova dados desnecessários após uso

// ✅ Contexto bem estruturado

@context {

customer\_id: String,

application: LoanApplication,

decision: Decision

}

// ❌ Contexto muito amplo

@context {

customer: FullCustomerProfile,

all\_applications: ApplicationHistory,

system\_state: CompleteSystemState

}

**3. Tratamento de Erros**

* **Específicidade**: Defina handlers específicos para erros conhecidos
* **Graceful Degradation**: Implemente fallbacks quando possível
* **Observabilidade**: Sempre registre erros para análise posterior

// ✅ Error handling completo

@error\_handling {

CreditBureau.timeout => use\_backup\_credit\_service,

CreditBureau.rate\_limit => wait\_and\_retry\_with\_backoff,

network\_error => retry\_with\_exponential\_backoff,

validation\_error => request\_data\_correction,

\* => log\_error\_and\_escalate\_to\_support

}

**Performance**

**1. Paralelização**

* Use parallel para operações independentes
* Implemente sync\_point para sincronização quando necessário
* Monitore overhead de paralelização

// ✅ Paralelização eficiente

parallel {

setup\_banking\_services,

setup\_digital\_access,

send\_welcome\_email

}

sync\_point account\_ready {

call NotificationService.accountActivated(customer)

}

**2. Timeouts Apropriados**

* Defina timeouts realistas baseados em SLAs
* Use timeouts diferentes para operações críticas vs. não-críticas
* Implemente escalação automática para timeouts

@config {

timeout: 24h, // Timeout geral do workflow

step\_timeout: 5m, // Timeout padrão por step

critical\_step\_timeout: 10m, // Timeout para steps críticos

network\_timeout: 30s // Timeout para chamadas de rede

}

**3. Otimização de Recursos**

* Use batching para operações de alto volume
* Implemente cache para dados frequentemente acessados
* Monitore uso de memória e CPU

**Segurança**

**1. Validação de Entrada**

* Sempre valide dados de entrada
* Use schemas para validação estruturada
* Sanitize dados antes de processamento

validate\_input {

call InputValidator.validate(input\_data, schema: "customer\_schema") -> validation

if validation.valid => process\_data

else => reject\_with\_error

}

**2. Controle de Acesso**

* Implemente autenticação para serviços críticos
* Use princípio de menor privilégio
* Registre todas as operações sensíveis

secure\_operation {

call AuthService.validateToken(request.token) -> auth\_result

if auth\_result.authorized => execute\_operation

else => return\_unauthorized\_error

}

**3. Auditoria**

* Mantenha trilha de auditoria completa
* Registre todas as decisões importantes
* Implemente retenção adequada de logs

audit\_decision {

call AuditLogger("loan\_decision", {

customer\_id: customer.id,

decision: decision.result,

amount: application.amount,

timestamp: now(),

decision\_factors: decision.factors

})

}

**Monitoramento**

**1. Métricas Essenciais**

* Taxa de sucesso por workflow
* Tempo de execução médio
* Throughput (workflows por hora)
* Taxa de erro por tipo

**2. Alertas Proativos**

* Configure alertas para degradação de performance
* Monitore filas de workflows pendentes
* Alerte para falhas de serviços críticos

**3. Dashboards**

* Crie dashboards para visibilidade operacional
* Implemente views específicas por equipe
* Mantenha histórico de métricas

**Troubleshooting**

**Problemas Comuns**

**1. Workflow Não Inicia**

**Sintomas:**

* Workflow fica em status "pending"
* Nenhum step é executado
* Sem logs de erro aparentes

**Causas Possíveis:**

* Dependências não registradas
* Contexto inicial inválido
* Triggers mal configurados

**Soluções:**

# Verificar dependências

runtime.check\_dependencies("WorkflowName")

# Validar contexto

compiler.validate\_context(initial\_context)

# Verificar triggers

runtime.list\_active\_triggers()

**2. Steps Ficam Pendentes**

**Sintomas:**

* Workflow para em step específico
* Timeout não é atingido
* Step não progride

**Causas Possíveis:**

* Condições never met
* Serviço externo não responde
* Evento aguardado nunca emitido

**Soluções:**

# Debug step específico

debugger.set\_breakpoint("WorkflowName", "step\_name")

debugger.inspect\_step\_state(instance\_id, "step\_name")

# Verificar eventos pendentes

runtime.list\_pending\_events(instance\_id)

# Forçar transição manual

runtime.force\_step\_transition(instance\_id, "next\_step")

**3. Erros de Plugin**

**Sintomas:**

* Exceções durante execução de plugin
* Plugin não é encontrado
* Configuração de plugin inválida

**Soluções:**

# Verificar plugins registrados

runtime.list\_plugins()

# Validar configuração do plugin

plugin.validate\_config(config)

# Recarregar plugin

runtime.reload\_plugin("PluginName")

**4. Performance Degradada**

**Sintomas:**

* Workflows executam mais lentamente
* Timeouts frequentes
* Alto uso de recursos

**Soluções:**

# Analisar métricas de performance

performance\_report = runtime.get\_performance\_metrics()

# Identificar gargalos

bottlenecks = analyzer.identify\_bottlenecks(performance\_report)

# Otimizar configuração

runtime.update\_config({

"max\_concurrent\_workflows": 50, # Reduzir concorrência

"step\_timeout": 300, # Aumentar timeout

"batch\_size": 100 # Ajustar batch size

})

**Ferramentas de Diagnóstico**

**1. Health Check**

def health\_check(runtime: FlowScriptRuntime) -> Dict[str, Any]:

"""Verifica saúde geral do sistema"""

return {

"runtime\_status": runtime.get\_status(),

"active\_workflows": len(runtime.running\_workflows),

"plugin\_status": runtime.check\_plugin\_health(),

"service\_status": runtime.check\_service\_health(),

"memory\_usage": runtime.get\_memory\_usage(),

"error\_rate": runtime.get\_error\_rate()

}

**2. Workflow Inspector**

def inspect\_workflow(runtime: FlowScriptRuntime, instance\_id: str) -> Dict[str, Any]:

"""Inspeciona estado detalhado de workflow"""

state = runtime.get\_workflow\_status(instance\_id)

return {

"current\_step": state.get("current\_step"),

"execution\_history": state.get("step\_history", []),

"context\_variables": state.get("context", {}),

"runtime\_variables": state.get("variables", {}),

"pending\_events": runtime.get\_pending\_events(instance\_id),

"error\_history": state.get("errors", [])

}

**3. Performance Profiler**

def profile\_workflow(workflow\_name: str, sample\_size: int = 100) -> Dict[str, Any]:

"""Profila performance de workflow"""

results = []

for i in range(sample\_size):

start\_time = time.time()

instance\_id = await runtime.start\_workflow(workflow\_name, sample\_context)

await runtime.wait\_for\_completion(instance\_id)

end\_time = time.time()

results.append({

"execution\_time": end\_time - start\_time,

"success": runtime.get\_workflow\_status(instance\_id)["status"] == "completed"

})

return analyze\_performance\_results(results)

**Logs e Debugging**

**1. Configuração de Logs**

import logging

# Configurar logging para FlowScript

logging.basicConfig(

level=logging.DEBUG,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler('flowscript.log'),

logging.StreamHandler()

]

)

# Logger específico para workflows

workflow\_logger = logging.getLogger('flowscript.workflows')

workflow\_logger.setLevel(logging.INFO)

**2. Debug Mode**

# Habilitar modo debug

runtime = FlowScriptRuntime(config={

"debug\_mode": True,

"trace\_execution": True,

"log\_level": "DEBUG"

})

# Debug específico de workflow

debugger = FlowScriptDebugger(runtime)

debugger.enable\_step\_mode()

debugger.set\_breakpoint("LoanApproval", "decision\_point")

**3. Análise de Traces**

def analyze\_trace(trace\_data: Dict) -> Dict[str, Any]:

"""Analisa trace de execução"""

spans = trace\_data.get("spans", [])

analysis = {

"total\_duration": trace\_data.get("duration", 0),

"step\_count": len(spans),

"slowest\_steps": sorted(spans, key=lambda x: x.get("duration", 0), reverse=True)[:5],

"error\_steps": [s for s in spans if s.get("status") == "error"],

"bottlenecks": identify\_bottlenecks(spans)

}

return analysis

**Conclusão**

O **FlowScript** representa uma evolução significativa na criação de workflows inteligentes, oferecendo:

**Principais Benefícios**

1. **Sintaxe Intuitiva**: Linguagem natural que facilita desenvolvimento e manutenção
2. **IA Integrada**: Otimização e análise automática por inteligência artificial
3. **Extensibilidade**: Sistema robusto de plugins e serviços
4. **Observabilidade**: Monitoramento completo e debugging avançado
5. **Escalabilidade**: Arquitetura preparada para alta concorrência

**Próximos Passos**

1. **Instalação**: Siga as instruções de instalação e configure seu ambiente
2. **Primeiro Workflow**: Crie um workflow simples usando os exemplos fornecidos
3. **Plugins**: Desenvolva plugins customizados para suas necessidades específicas
4. **Produção**: Configure monitoramento e implemente em ambiente produtivo
5. **Otimização**: Use ferramentas de análise para otimizar performance

**Recursos Adicionais**

* **Comunidade**: Participe da comunidade FlowScript para compartilhar experiências
* **Documentação**: Consulte documentação adicional para casos de uso específicos
* **Suporte**: Entre em contato para suporte técnico e consultoria

O FlowScript está preparado para revolucionar como você cria e gerencia workflows complexos, combinando simplicidade de uso com poder computacional avançado.

*Documentação FlowScript v1.0 - © 2024 FlowScript Project*# FlowScript - Documentação Completa

**Índice**

1. [Introdução](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#introdu%C3%A7%C3%A3o)
2. [Instalação e Dependências](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#instala%C3%A7%C3%A3o-e-depend%C3%AAncias)
3. [Sintaxe da Linguagem](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#sintaxe-da-linguagem)
4. [Compilação e Execução](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#compila%C3%A7%C3%A3o-e-execu%C3%A7%C3%A3o)
5. [Sistema de Plugins](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#sistema-de-plugins)
6. [Integração com Serviços](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#integra%C3%A7%C3%A3o-com-servi%C3%A7os)
7. [Tratamento de Erros](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#tratamento-de-erros)
8. [Monitoramento e Debugging](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#monitoramento-e-debugging)
9. [Exemplos Práticos](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#exemplos-pr%C3%A1ticos)
10. [API Reference](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#api-reference)
11. [Melhores Práticas](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#melhores-pr%C3%A1ticas)
12. [Troubleshooting](https://claude.ai/chat/f1382298-de2b-45aa-ad80-6ce99d40a609#troubleshooting)

**Introdução**

**FlowScript** é uma linguagem domain-specific (DSL) projetada especificamente para criação de workflows baseados em eventos com integração nativa de IA. A linguagem combina sintaxe natural intuitiva com capacidades de execução assíncrona e otimização automática.

**Características Principais**

* **Sintaxe Natural**: Próxima à linguagem humana para facilitar compreensão por IA
* **Eventos Assíncronos**: Sistema de eventos de primeira classe
* **Plugins Dinâmicos**: Extensibilidade através de plugins personalizados
* **IA Integrada**: Otimização e análise automática por inteligência artificial
* **Error Handling**: Tratamento robusto de erros com estratégias de recuperação
* **Monitoramento**: Observabilidade completa de execução

**Casos de Uso**

* Aprovação de empréstimos e processos financeiros
* Onboarding de clientes
* Processamento de documentos
* Workflows de aprovação empresarial
* Automação de processos de negócio
* Pipelines de dados com IA

**Instalação e Dependências**

**Requisitos do Sistema**

* **Python**: 3.8 ou superior
* **Sistema Operacional**: Linux, macOS, Windows
* **Memória**: Mínimo 2GB RAM
* **Armazenamento**: 100MB de espaço livre

**Dependências Python**

# Dependências obrigatórias

pip install asyncio dataclasses typing

# Dependências opcionais (para funcionalidades avançadas)

pip install aiohttp # Para integração HTTP

pip install aioredis # Para cache Redis

pip install sqlalchemy # Para integração com banco de dados

pip install pydantic # Para validação de schemas

pip install prometheus-client # Para métricas

**Instalação do FlowScript**

# Clonar repositório

git clone https://github.com/yourorg/flowscript.git

cd flowscript

# Instalar dependências

pip install -r requirements.txt

# Instalação do pacote

pip install -e .

**Estrutura de Diretórios**

flowscript/

├── src/

│ ├── flowscript/

│ │ ├── lexer.py # Analisador léxico

│ │ ├── parser.py # Parser da linguagem

│ │ ├── runtime.py # Runtime de execução

│ │ ├── compiler.py # Compilador

│ │ ├── plugins/ # Plugins built-in

│ │ ├── templates/ # Templates de workflow

│ │ └── tools/ # Ferramentas de desenvolvimento

│ └── tests/ # Testes unitários

├── examples/ # Exemplos de uso

├── docs/ # Documentação adicional

├── requirements.txt # Dependências

└── setup.py # Configuração de instalação

**Verificação da Instalação**

# test\_installation.py

from flowscript import FlowScriptCompiler, FlowScriptRuntime

def test\_installation():

"""Testa se FlowScript foi instalado corretamente"""

# Teste básico de compilação

code = """

@workflow TestWorkflow {

@context {

message: String

}

@flow {

start {

notify message

}

}

}

"""

try:

compiler = FlowScriptCompiler()

workflow = compiler.compile(code)

print("✅ FlowScript instalado com sucesso!")

print(f"Workflow compilado: {workflow.name}")

return True

except Exception as e:

print(f"❌ Erro na instalação: {e}")

return False

if \_\_name\_\_ == "\_\_main\_\_":

test\_installation()

**Sintaxe da Linguagem**

**Estrutura Básica**

Todo workflow FlowScript segue a estrutura fundamental:

@workflow NomeDoWorkflow {

@context {

// Definição de variáveis globais

}

@triggers {

// Mapeamento de eventos para ações

}

@flow {

// Definição dos steps do workflow

}

@error\_handling {

// Tratamento de erros

}

@config {

// Configurações do workflow

}

}

**Seção @context**

Define variáveis e tipos disponíveis globalmente no workflow:

@context {

customer: Customer, // Objeto customer

application: LoanApplication, // Aplicação de empréstimo

score: Number, // Score numérico

approved: Boolean, // Flag booleano

documents: DocumentList, // Lista de documentos

metadata: JsonObject // Objeto JSON genérico

}

**Tipos Suportados:**

* String: Texto
* Number: Números inteiros ou decimais
* Boolean: true/false
* Duration: Tempo (30s, 5m, 2h)
* JsonObject: Objeto JSON genérico
* Array: Lista de elementos
* Tipos customizados (Customer, LoanApplication, etc.)

**Seção @triggers**

Mapeia eventos para steps iniciais:

@triggers {

application\_received => validate\_data,

credit\_check\_complete => decision\_point,

user\_input\_received => process\_input,

timer\_expired => cleanup\_workflow,

external\_api\_callback => handle\_callback

}

**Tipos de Eventos:**

* **Eventos de Sistema**: workflow\_started, timeout\_reached
* **Eventos Customizados**: Definidos pelo desenvolvedor
* **Eventos Externos**: Vindos de APIs, webhooks, etc.
* **Eventos Temporais**: Baseados em tempo ou cronogramas

**Seção @flow**

Define a lógica principal do workflow através de steps:

@flow {

step\_name {

// Ações sequenciais

action1

action2

// Condicionais

if condition => next\_step

else => alternative\_step

// Branches múltiplos

branch {

condition1 => action1,

condition2 => action2,

condition3 => action3

}

// Chamadas de serviço

call Service.method(params) -> result\_var

// Emissão de eventos

emit event\_name

// Aguardar eventos

wait\_for event\_name

}

}

**Condicionais**

**Condicionais Simples**

if score > 700 => approve\_automatically

else => manual\_review

**Condicionais Compostas**

if score > 700 && risk\_level == "low" => premium\_approval

else if score > 600 => standard\_approval

else => detailed\_review

**Branch Statements**

branch {

score >= 750 && risk == "low" => auto\_approve\_premium,

score >= 650 && risk != "high" => auto\_approve\_standard,

score >= 550 => manual\_review,

score < 550 => auto\_reject

}

**Operadores**

**Operadores de Comparação**

* ==: Igual
* !=: Diferente
* >: Maior que
* <: Menor que
* >=: Maior ou igual
* <=: Menor ou igual

**Operadores Lógicos**

* &&: E lógico
* ||: OU lógico
* !: NÃO lógico

**Operadores de Atribuição**

* ->: Atribui resultado de chamada
* =: Atribuição simples

**Chamadas de Serviço e Plugins**

// Chamada de serviço externo

call CreditBureau.getScore(customer.ssn) -> credit\_score

// Chamada de plugin

call EmailSender(customer.email, "Subject", "Body") -> email\_result

// Chamada com múltiplos parâmetros

call RiskAnalyzer.analyze(customer, application, credit\_score) -> risk\_assessment

// Chamada sem retorno

call NotificationService.send("Message", customer.phone)

**Ações Built-in**

// Verificações

check application.completeness

check customer.eligibility

// Aprovações/Rejeições

approve application

reject application with reason("Insufficient income")

// Notificações

notify customer "Your application is being processed"

notify admin "Manual review required"

// Roteamento

route\_to queue\_name

route\_to analyst\_queue with priority("high")

// Aguardar

wait\_for user\_response

wait\_for external\_callback

**Seção @error\_handling**

Define como tratar diferentes tipos de erros:

@error\_handling {

// Erro específico de serviço

CreditBureau.timeout => use\_backup\_credit\_service,

// Erro de categoria

network\_error => retry\_with\_backoff,

// Erro genérico por serviço

PaymentService.\* => escalate\_to\_finance\_team,

// Handler global

\* => log\_error\_and\_continue,

// Múltiplas ações

critical\_system\_error => {

notify admin "Critical error occurred",

escalate\_to\_engineering,

pause\_workflow

}

}

**Seção @config**

Configurações do workflow:

@config {

// Timeouts

timeout: 30m,

step\_timeout: 5m,

// Retry policies

max\_retries: 3,

retry\_policy: exponential\_backoff,

// Prioridades

default\_priority: medium,

escalation\_timeout: 24h,

// IA e otimização

ai\_optimization: true,

learning\_mode: active,

// Monitoramento

audit\_all\_steps: true,

metrics\_enabled: true,

// Concorrência

max\_concurrent\_instances: 100,

batch\_size: 10

}

**Compilação e Execução**

**Processo de Compilação**

O FlowScript passa por várias etapas de compilação:

Código FlowScript (.fs)

↓

[Lexer] → Tokens

↓

[Parser] → AST (Abstract Syntax Tree)

↓

[Semantic Analyzer] → Validação e otimização

↓

[Code Generator] → Bytecode executável

↓

[Runtime] → Execução

**Compilador FlowScript**

from flowscript import FlowScriptCompiler

# Inicializar compilador

compiler = FlowScriptCompiler()

# Compilar de arquivo

workflow = compiler.compile\_file("workflow.fs")

# Compilar de string

code = """

@workflow Example {

@flow {

start {

notify "Hello World"

}

}

}

"""

workflow = compiler.compile(code)

# Compilar com opções

workflow = compiler.compile(code, options={

"optimize": True,

"debug": False,

"strict\_mode": True

})

**Opções de Compilação**

compile\_options = {

"optimize": True, # Habilita otimizações

"debug": False, # Modo debug

"strict\_mode": True, # Validação rigorosa

"target\_version": "1.0", # Versão alvo

"warnings\_as\_errors": False, # Trata warnings como erros

"generate\_docs": True, # Gera documentação automática

"validate\_plugins": True # Valida plugins referenciados

}

**Runtime de Execução**

from flowscript import FlowScriptRuntime

# Inicializar runtime

runtime = FlowScriptRuntime()

# Carregar workflow compilado

runtime.load\_workflow(workflow)

# Registrar serviços

runtime.register\_service("CreditBureau", credit\_bureau\_service)

runtime.register\_service("EmailService", email\_service)

# Executar workflow

instance\_id = await runtime.start\_workflow(

workflow\_name="LoanApproval",

context={

"customer": customer\_data,

"application": loan\_application

}

)

# Monitorar execução

status = runtime.get\_workflow\_status(instance\_id)

print(f"Status: {status}")

**Configuração do Runtime**

runtime\_config = {

"max\_concurrent\_workflows": 1000,

"event\_queue\_size": 10000,

"step\_timeout\_default": 300, # 5 minutos

"retry\_attempts": 3,

"metrics\_enabled": True,

"audit\_enabled": True,

"plugin\_sandbox": True

}

runtime = FlowScriptRuntime(config=runtime\_config)

**Execução de Workflows**

**Execução Síncrona**

# Para workflows simples que completam rapidamente

result = runtime.execute\_workflow\_sync("WorkflowName", context)

**Execução Assíncrona**

# Para workflows longos ou com esperas

instance\_id = await runtime.start\_workflow("WorkflowName", context)

# Aguardar conclusão

result = await runtime.wait\_for\_completion(instance\_id)

**Execução em Lote**

# Para processar múltiplos workflows

batch\_results = await runtime.execute\_batch([

{"workflow": "Approval", "context": context1},

{"workflow": "Approval", "context": context2},

{"workflow": "Approval", "context": context3}

])

**Sistema de Plugins**

**Arquitetura de Plugins**

Os plugins no FlowScript são componentes modulares que estendem a funcionalidade do sistema. Cada plugin implementa uma interface padronizada e pode ser carregado dinamicamente.

**Interface de Plugin**

from abc import ABC, abstractmethod

from typing import Any, Dict, List, Optional

class FlowScriptPlugin(ABC):

"""Interface base para plugins FlowScript"""

@property

@abstractmethod

def name(self) -> str:

"""Nome único do plugin"""

pass

@property

@abstractmethod

def version(self) -> str:

"""Versão do plugin"""

pass

@property

@abstractmethod

def description(self) -> str:

"""Descrição do plugin"""

pass

@property

def dependencies(self) -> List[str]:

"""Lista de dependências (outros plugins ou serviços)"""

return []

@property

def parameters(self) -> List[Dict[str, Any]]:

"""Schema dos parâmetros aceitos"""

return []

@abstractmethod

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa o plugin"""

pass

@abstractmethod

async def execute(self, \*args, \*\*kwargs) -> Any:

"""Executa a funcionalidade principal do plugin"""

pass

async def cleanup(self) -> None:

"""Limpeza de recursos (opcional)"""

pass

def validate\_parameters(self, \*args, \*\*kwargs) -> bool:

"""Valida parâmetros de entrada (opcional)"""

return True

**Criando um Plugin Simples**

from flowscript.plugins import FlowScriptPlugin

import aiohttp

import asyncio

class EmailSenderPlugin(FlowScriptPlugin):

"""Plugin para envio de emails"""

@property

def name(self) -> str:

return "EmailSender"

@property

def version(self) -> str:

return "1.0.0"

@property

def description(self) -> str:

return "Envia emails através de serviços SMTP ou APIs"

@property

def parameters(self) -> List[Dict[str, Any]]:

return [

{

"name": "recipient",

"type": "string",

"required": True,

"description": "Email do destinatário"

},

{

"name": "subject",

"type": "string",

"required": True,

"description": "Assunto do email"

},

{

"name": "body",

"type": "string",

"required": True,

"description": "Corpo do email"

},

{

"name": "template",

"type": "string",

"required": False,

"description": "Template a ser usado"

}

]

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa configurações do email"""

self.smtp\_server = config.get("smtp\_server", "localhost")

self.smtp\_port = config.get("smtp\_port", 587)

self.username = config.get("username")

self.password = config.get("password")

self.api\_key = config.get("api\_key")

self.service\_type = config.get("service\_type", "smtp") # smtp, sendgrid, ses

print(f"EmailSender plugin inicializado ({self.service\_type})")

async def execute(self, recipient: str, subject: str, body: str,

template: Optional[str] = None) -> Dict[str, Any]:

"""Envia email"""

# Validar entrada

if not self.validate\_parameters(recipient, subject, body):

raise ValueError("Parâmetros inválidos para envio de email")

# Aplicar template se especificado

if template:

body = await self.\_apply\_template(template, body)

# Enviar baseado no tipo de serviço

if self.service\_type == "smtp":

return await self.\_send\_via\_smtp(recipient, subject, body)

elif self.service\_type == "sendgrid":

return await self.\_send\_via\_sendgrid(recipient, subject, body)

elif self.service\_type == "ses":

return await self.\_send\_via\_ses(recipient, subject, body)

else:

raise ValueError(f"Tipo de serviço não suportado: {self.service\_type}")

def validate\_parameters(self, recipient: str, subject: str, body: str) -> bool:

"""Valida parâmetros de email"""

import re

# Validar formato do email

email\_pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

if not re.match(email\_pattern, recipient):

return False

# Validar campos obrigatórios

if not subject.strip() or not body.strip():

return False

return True

async def \_apply\_template(self, template: str, data: str) -> str:

"""Aplica template ao corpo do email"""

# Implementação simplificada - em produção usar Jinja2 ou similar

templates = {

"welcome": "Bem-vindo! {data}",

"approval": "Parabéns! Sua solicitação foi aprovada. {data}",

"rejection": "Lamentamos informar que sua solicitação foi rejeitada. {data}"

}

template\_body = templates.get(template, "{data}")

return template\_body.format(data=data)

async def \_send\_via\_smtp(self, recipient: str, subject: str, body: str) -> Dict[str, Any]:

"""Envia email via SMTP"""

# Simula envio SMTP

await asyncio.sleep(0.1) # Simula latência

print(f"📧 SMTP: Enviando email para {recipient}")

return {

"status": "sent",

"message\_id": f"smtp\_{int(asyncio.get\_event\_loop().time())}",

"service": "smtp"

}

async def \_send\_via\_sendgrid(self, recipient: str, subject: str, body: str) -> Dict[str, Any]:

"""Envia email via SendGrid API"""

# Implementação real usaria SendGrid API

await asyncio.sleep(0.2) # Simula latência da API

print(f"📧 SendGrid: Enviando email para {recipient}")

return {

"status": "sent",

"message\_id": f"sg\_{int(asyncio.get\_event\_loop().time())}",

"service": "sendgrid"

}

async def \_send\_via\_ses(self, recipient: str, subject: str, body: str) -> Dict[str, Any]:

"""Envia email via AWS SES"""

# Implementação real usaria boto3

await asyncio.sleep(0.15) # Simula latência da AWS

print(f"📧 AWS SES: Enviando email para {recipient}")

return {

"status": "sent",

"message\_id": f"ses\_{int(asyncio.get\_event\_loop().time())}",

"service": "aws\_ses"

}

**Plugin de Integração com API**

class APIConnectorPlugin(FlowScriptPlugin):

"""Plugin genérico para chamadas de API"""

@property

def name(self) -> str:

return "APIConnector"

@property

def version(self) -> str:

return "1.0.0"

@property

def description(self) -> str:

return "Conecta com APIs REST de forma genérica"

@property

def parameters(self) -> List[Dict[str, Any]]:

return [

{"name": "url", "type": "string", "required": True},

{"name": "method", "type": "string", "required": False, "default": "GET"},

{"name": "headers", "type": "object", "required": False},

{"name": "payload", "type": "object", "required": False},

{"name": "timeout", "type": "number", "required": False, "default": 30}

]

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa cliente HTTP"""

self.session = aiohttp.ClientSession(

timeout=aiohttp.ClientTimeout(total=config.get("default\_timeout", 30))

)

self.default\_headers = config.get("default\_headers", {})

async def execute(self, url: str, method: str = "GET",

headers: Optional[Dict] = None,

payload: Optional[Dict] = None,

timeout: int = 30) -> Dict[str, Any]:

"""Executa chamada de API"""

# Monta headers

request\_headers = {\*\*self.default\_headers}

if headers:

request\_headers.update(headers)

try:

async with self.session.request(

method=method.upper(),

url=url,

headers=request\_headers,

json=payload,

timeout=aiohttp.ClientTimeout(total=timeout)

) as response:

result = {

"status\_code": response.status,

"headers": dict(response.headers),

"success": 200 <= response.status < 300

}

# Tenta parsear JSON, senão retorna texto

try:

result["data"] = await response.json()

except:

result["data"] = await response.text()

return result

except asyncio.TimeoutError:

raise Exception(f"Timeout na chamada para {url}")

except Exception as e:

raise Exception(f"Erro na chamada de API: {str(e)}")

async def cleanup(self) -> None:

"""Fecha sessão HTTP"""

if self.session:

await self.session.close()

**Plugin de Processamento de Dados**

class DataProcessorPlugin(FlowScriptPlugin):

"""Plugin para processamento de dados"""

@property

def name(self) -> str:

return "DataProcessor"

@property

def version(self) -> str:

return "1.0.0"

@property

def description(self) -> str:

return "Processa e transforma dados"

async def initialize(self, config: Dict[str, Any]) -> None:

"""Inicializa processador"""

self.transformations = config.get("transformations", {})

self.validators = config.get("validators", {})

async def execute(self, data: Any, operation: str, \*\*kwargs) -> Any:

"""Executa operação nos dados"""

if operation == "validate":

return await self.\_validate\_data(data, kwargs.get("schema"))

elif operation == "transform":

return await self.\_transform\_data(data, kwargs.get("transformation"))

elif operation == "aggregate":

return await self.\_aggregate\_data(data, kwargs.get("aggregation"))

elif operation == "filter":

return await self.\_filter\_data(data, kwargs.get("criteria"))

else:

raise ValueError(f"Operação não suportada: {operation}")

async def \_validate\_data(self, data: Any, schema: Dict) -> Dict[str, Any]:

"""Valida dados contra schema"""

errors = []

# Implementação simplificada de validação

if isinstance(data, dict) and isinstance(schema, dict):

for field, field\_schema in schema.items():

if field\_schema.get("required", False) and field not in data:

errors.append(f"Campo obrigatório '{field}' ausente")

if field in data:

expected\_type = field\_schema.get("type")

if expected\_type == "string" and not isinstance(data[field], str):

errors.append(f"Campo '{field}' deve ser string")

elif expected\_type == "number" and not isinstance(data[field], (int, float)):

errors.append(f"Campo '{field}' deve ser número")

return {

"valid": len(errors) == 0,

"errors": errors,

"data": data

}

async def \_transform\_data(self, data: Any, transformation: str) -> Any:

"""Transforma dados"""

transformations = {

"uppercase": lambda x: x.upper() if isinstance(x, str) else x,

"lowercase": lambda x: x.lower() if isinstance(x, str) else x,

"strip": lambda x: x.strip() if isinstance(x, str) else x,

"round": lambda x: round(x, 2) if isinstance(x, (int, float)) else x

}

transform\_func = transformations.get(transformation)

if not transform\_func:

raise ValueError(f"Transformação não encontrada: {transformation}")

if isinstance(data, list):

return [transform\_func(item) for item in data]

elif isinstance(data, dict):

return {k: transform\_func(v) for k, v in data.items()}

else:

return transform\_func(data)

async def \_aggregate\_data(self, data: List, aggregation: str) -> Any:

"""Agrega dados"""

if not isinstance(data, list):

raise ValueError("Agregação requer lista de dados")

if aggregation == "sum":

return sum(x for x in data if isinstance(x, (int, float)))

elif aggregation == "avg":

numbers = [x for x in data if isinstance(x, (int, float))]

return sum(numbers) / len(numbers) if numbers else 0

elif aggregation == "count":

return len(data)

elif aggregation == "max":

numbers = [x for x in data if isinstance(x, (int, float))]

return max(numbers) if numbers else None

elif aggregation == "min":

numbers = [x for x in data if isinstance(x, (int, float))]

return min(numbers) if numbers else None

else:

raise ValueError(f"Agregação não suportada: {aggregation}")

async def \_filter\_data(self, data: List, criteria: Dict) -> List:

"""Filtra dados baseado em critérios"""

if not isinstance(data, list):

raise ValueError("Filtro requer lista de dados")

filtered = []

for item in data:

if self.\_matches\_criteria(item, criteria):

filtered.append(item)

return filtered

def \_matches\_criteria(self, item: Any, criteria: Dict) -> bool:

"""Verifica se item atende aos critérios"""

if not isinstance(item, dict) or not isinstance(criteria, dict):

return True

for field, condition in criteria.items():

if field not in item:

return False

value = item[field]

if isinstance(condition, dict):

# Condições complexas como {"gt": 100, "lt": 200}

for op, expected in condition.items():

if op == "gt" and not (value > expected):

return False

elif op == "lt" and not (value < expected):

return False

elif op == "eq" and not (value == expected):

return False

elif op == "ne" and not (value != expected):

return False

else:

# Condição simples de igualdade

if value != condition:

return False

return True

**Registrando Plugins**

from flowscript import FlowScriptRuntime

# Inicializar runtime

runtime = FlowScriptRuntime()

# Registrar plugin individual

email\_plugin = EmailSenderPlugin()

await email\_plugin.initialize({

"service\_type": "sendgrid",

"api\_key": "your\_api\_key"

})

runtime.register\_plugin(email\_plugin)

# Registrar múltiplos plugins

plugins = [

EmailSenderPlugin(),

APIConnectorPlugin(),

DataProcessorPlugin()

]

for plugin in plugins:

await plugin.initialize({})

runtime.register\_plugin(plugin)

# Registrar plugin com configuração

runtime.register\_plugin\_with\_config("DatabaseConnector", {

"connection\_string": "postgresql://user:pass@localhost/db",

"pool\_size": 10,

"timeout": 30

})

# Carregar plugins de diretório

runtime.load\_plugins\_from\_directory("./plugins/")

# Listar plugins registrados

available\_plugins = runtime.list\_plugins()

print("Plugins disponíveis:", available\_plugins)

**Plugin Loader e Discovery**

import os

import importlib.util

from typing import List, Type

class PluginLoader:

"""Carregador automático de plugins"""

def \_\_init\_\_(self, plugin\_directories: List[str]):

self.plugin\_directories = plugin\_directories

self.loaded\_plugins: Dict[str, FlowScriptPlugin] = {}

async def discover\_plugins(self) -> List[Type[FlowScriptPlugin]]:

"""Descobre plugins nos diretórios especificados"""

plugin\_classes = []

for directory in self.plugin\_directories:

if not os.path.exists(directory):

continue

for filename in os.listdir(directory):

if filename.endswith('.py') and not filename.startswith('\_\_'):

plugin\_path = os.path.join(directory, filename)

plugin\_classes.extend(self.\_load\_plugins\_from\_file(plugin\_path))

return plugin\_classes

def \_load\_plugins\_from\_file(self, file\_path: str) -> List[Type[FlowScriptPlugin]]:

"""Carrega plugins de um arquivo Python"""

spec = importlib.util.spec\_from\_file\_location("plugin\_module", file\_path)

module = importlib.util.module\_from\_spec(spec)

spec.loader.exec\_module(module)

plugin\_classes = []

for attr\_name in dir(module):

attr = getattr(module, attr\_name)

if (isinstance(attr, type) and

issubclass(attr, FlowScriptPlugin) and

attr != FlowScriptPlugin):

plugin\_classes.append(attr)

return plugin\_classes

async def load\_all\_plugins(self, runtime: FlowScriptRuntime,

config: Dict[str, Dict] = None) -> None:

"""Carrega todos os plugins descobertos"""

plugin\_classes = await self.discover\_plugins()

for plugin\_class in plugin\_classes:

try:

plugin = plugin\_class()

plugin\_config = config.get(plugin.name, {}) if config else {}

await plugin.initialize(plugin\_config)

runtime.register\_plugin(plugin)

self.loaded\_plugins[plugin.name] = plugin

print(f"✅ Plugin '{plugin.name}' carregado com sucesso")

except Exception as e:

print(f"❌ Erro ao carregar plugin '{plugin\_class.\_\_name\_\_}': {e}")

# Exemplo de uso

loader = PluginLoader(["./plugins", "./custom\_plugins"])

await loader.load\_all\_plugins(runtime, {

"EmailSender": {"service\_type": "sendgrid", "api\_key": "key"},

"DatabaseConnector": {"connection\_string": "sqlite:///app.db"}

})

**Integração com Serviços**

**Sistema de Serviços**

O FlowScript distingue entre **plugins** (componentes internos) e **serviços** (sistemas externos). Serviços são registrados separadamente e podem representar APIs, bancos de dados, sistemas de mensageria, etc.

**Criando um Serviço**

from typing import Any, Dict, List, Optional

import asyncio

import aiohttp

class CreditBureauService:

"""Serviço para integração com bureau de crédito"""

def \_\_init\_\_(self, api\_url: str, api\_key: str, timeout: int = 30):

self.api\_url = api\_url

self.api\_key = api\_key

self.timeout = timeout

self.session = None

async def initialize(self):

"""Inicializa conexão com serviço"""

self.session = aiohttp.ClientSession(

timeout=aiohttp.ClientTimeout(total=self.timeout),

headers={"Authorization": f"Bearer {self.api\_key}"}

)

print("CreditBureau service initialized")

async def getScore(self, ssn: str) -> int:

"""Obtém score de crédito"""

url = f"{self.api\_url}/credit-score"

payload = {"ssn": ssn}

try:

async with self.session.post(url, json=payload) as response:

if response.status == 200:

data = await response.json()

return data.get("score", 0)

else:

raise Exception(f"API returned status {response.status}")

except asyncio.TimeoutError:

raise Exception("Credit bureau timeout")

except Exception as e:

raise Exception(f"Credit bureau error: {str(e)}")

async def getFullReport(self, ssn: str) -> Dict[str, Any]:

"""Obtém relatório completo de crédito"""

url = f"{self.api\_url}/full-report"

payload = {"ssn": ssn}

async with self.session.post(url, json=payload) as response:

return await response.json()

async def close(self):

"""Fecha conexões"""

if self.session:

await self.session.close()

class DatabaseService:

"""Serviço genérico para banco de dados"""

def \_\_init\_\_(self, connection\_string: str):

self.connection\_string = connection\_string

self.pool = None

async def initialize(self):

"""Inicializa pool de conexões"""

# Em implementação real, usaria SQLAlchemy ou asyncpg

print(f"Database service initialized: {self.connection\_string}")

self.pool = "mock\_pool" # Mock para exemplo

async def execute\_query(self, query: str, parameters: List = None) -> List[Dict]:

"""Executa query SELECT"""

print(f"Executing query: {query}")

if parameters:

print(f"Parameters: {parameters}")

# Mock result - em produção, executaria query real

return [{"id": 1, "result": "mock\_data"}]

async def execute\_command(self, command: str, parameters: List = None) -> int:

"""Executa comando INSERT/UPDATE/DELETE"""

print(f"Executing command: {command}")

if parameters:

print(f"Parameters: {parameters}")

# Mock - retorna número de linhas afetadas

return 1

async def close(self):

"""Fecha pool de conexões"""

print("Database connections closed")

class MessageQueueService:

"""Serviço para sistema de mensageria"""

def \_\_init\_\_(self, broker\_url: str):

self.broker\_url = broker\_url

self.connection = None

async def initialize(self):

"""Conecta ao broker"""

print(f"MessageQueue service initialized: {self.broker\_url}")

self.connection = "mock\_connection"

async def publish(self, queue: str, message: Dict[str, Any]) -> bool:

"""Publica mensagem em fila"""

print(f"Publishing to {queue}: {message}")

await asyncio.sleep(0.1) # Simula latência

return True

async def subscribe(self, queue: str, callback: callable) -> None:

"""Inscreve-se em fila para receber mensagens"""

print(f"Subscribing to queue: {queue}")

# Em implementação real, configuraria callback

async def close(self):

"""Fecha conexão"""

print("MessageQueue connection closed")

**Registrando Serviços**

from flowscript import FlowScriptRuntime

async def setup\_services(runtime: FlowScriptRuntime):

"""Configura todos os serviços necessários"""

# Criar e inicializar serviços

credit\_bureau = CreditBureauService(

api\_url="https://api.creditbureau.com",

api\_key="your\_api\_key"

)

await credit\_bureau.initialize()

database = DatabaseService("postgresql://user:pass@localhost/flowscript")

await database.initialize()

message\_queue = MessageQueueService("redis://localhost:6379")

await message\_queue.initialize()

# Registrar no runtime

runtime.register\_service("CreditBureau", credit\_bureau)

runtime.register\_service("Database", database)

runtime.register\_service("MessageQueue", message\_queue)

print("✅ Todos os serviços registrados")

# Usar em workflow

runtime = FlowScriptRuntime()

await setup\_services(runtime)

**Service Discovery e Health Checks**

class ServiceRegistry:

"""Registry para descoberta e monitoramento de serviços"""

def \_\_init\_\_(self):

self.services: Dict[str, Any] = {}

self.health\_status: Dict[str, bool] = {}

self.last\_health\_check: Dict[str, float] = {}

def register\_service(self, name: str, service: Any, health\_check\_interval: int = 60):

"""Registra serviço com health check"""

self.services[name] = service

self.health\_status[name] = True

self.last\_health\_check[name] = 0

# Agendar health checks periódicos

asyncio.create\_task(self.\_periodic\_health\_check(name, health\_check\_interval))

async def \_periodic\_health\_check(self, service\_name: str, interval: int):

"""Executa health check periódico"""

while service\_name in self.services:

try:

service = self.services[service\_name]

if hasattr(service, 'health\_check'):

is\_healthy = await service.health\_check()

else:

is\_healthy = True # Assume saudável se não tem health check

self.health\_status[service\_name] = is\_healthy

self.last\_health\_check[service\_name] = asyncio.get\_event\_loop().time()

if not is\_healthy:

print(f"⚠️ Service {service\_name} is unhealthy")

except Exception as e:

print(f"❌ Health check failed for {service\_name}: {e}")

self.health\_status[service\_name] = False

await asyncio.sleep(interval)

def get\_healthy\_services(self) -> List[str]:

"""Retorna lista de serviços saudáveis"""

return [name for name, healthy in self.health\_status.items() if healthy]

def get\_service\_status(self) -> Dict[str, Dict]:

"""Retorna status completo de todos os serviços"""

status = {}

for name in self.services:

status[name] = {

"healthy": self.health\_status.get(name, False),

"last\_check": self.last\_health\_check.get(name, 0)

}

return status

# Integração com runtime

class EnhancedFlowScriptRuntime(FlowScriptRuntime):

"""Runtime com service discovery integrado"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.service\_registry = ServiceRegistry()

def register\_service(self, name: str, service: Any, health\_check\_interval: int = 60):

"""Registra serviço com health monitoring"""

super().register\_service(name, service)

self.service\_registry.register\_service(name, service, health\_check\_interval)

async def get\_service\_health(self) -> Dict[str, Dict]:

"""Retorna status de saúde dos serviços"""

return self.service\_registry.get\_service\_status()

**Tratamento de Erros**

**Estratégias de Error Handling**

O FlowScript oferece múltiplas estratégias para tratamento de erros, desde recuperação automática até escalação para intervenção humana.

**Tipos de Erros**

@error\_handling {

// 1. Erros específicos de serviço

CreditBureau.timeout => retry\_with\_backoff,

CreditBureau.rate\_limit => wait\_and\_retry,

CreditBureau.unauthorized => refresh\_credentials,

// 2. Erros por categoria

network\_error => switch\_to\_backup\_service,

validation\_error => request\_data\_correction,

authorization\_error => escalate\_to\_admin,

// 3. Erros por padrão

\*.timeout => retry\_operation,

\*.connection\_error => check\_service\_health,

// 4. Tratamento em cascata

critical\_system\_error => {

log\_critical\_error,

notify\_engineering\_team,

pause\_workflow,

escalate\_to\_on\_call

},

// 5. Fallback global

\* => log\_and\_continue\_with\_degraded\_service

}

**Políticas de Retry**

from typing import Callable, Any

import asyncio

import random

class RetryPolicy:

"""Políticas de retry para operações"""

@staticmethod

async def exponential\_backoff(func: Callable, max\_attempts: int = 3,

base\_delay: float = 1.0, max\_delay: float = 60.0):

"""Retry com backoff exponencial"""

for attempt in range(max\_attempts):

try:

return await func()

except Exception as e:

if attempt == max\_attempts - 1:

raise e

# Calcular delay com jitter

delay = min(base\_delay \* (2 \*\* attempt), max\_delay)

jitter = random.uniform(0, 0.1) \* delay

total\_delay = delay + jitter

print(f"Attempt {attempt + 1} failed, retrying in {total\_delay:.2f}s")

await asyncio.sleep(total\_delay)

@staticmethod

async def fixed\_interval(func: Callable, max\_attempts: int = 3, interval: float = 5.0):

"""Retry com intervalo fixo"""

for attempt in range(max\_attempts):

try:

return await func()

except Exception as e:

if attempt == max\_attempts - 1:

raise e

print(f"Attempt {attempt + 1} failed, retrying in {interval}s")

await asyncio.sleep(interval)

@staticmethod

async def linear\_backoff(func: Callable, max\_attempts: int = 3, base\_delay: float = 1.0):

"""Retry com backoff linear"""

for attempt in range(max\_attempts):

try:

return await func()

except Exception as e:

if attempt == max\_attempts - 1:

raise e

delay = base\_delay \* (attempt + 1)

print(f"Attempt {attempt + 1} failed, retrying in {delay}s")

await asyncio.sleep(delay)

class CircuitBreaker:

"""Implementa padrão Circuit Breaker"""

def \_\_init\_\_(self, failure\_threshold: int = 5, recovery\_timeout: int = 60):

self.failure\_threshold = failure\_threshold

self.recovery\_timeout = recovery\_timeout

self.failure\_count = 0

self.last\_failure\_time = 0

self.state = "CLOSED" # CLOSED, OPEN, HALF\_OPEN

async def call(self, func: Callable, \*args, \*\*kwargs):

"""Executa função com circuit breaker"""

if self.state == "OPEN":

if asyncio.get\_event\_loop().time() - self.last\_failure\_time > self.recovery\_timeout:

self.state = "HALF\_OPEN"

print("Circuit breaker transitioning to HALF\_OPEN")

else:

raise Exception("Circuit breaker is OPEN")

try:

result = await func(\*args, \*\*kwargs)

if self.state == "HALF\_OPEN":

self.state = "CLOSED"

self.failure\_count = 0

print("Circuit breaker CLOSED - service recovered")

return result

except Exception as e:

self.failure\_count += 1

self.last\_failure\_time = asyncio.get\_event\_loop().time()

if self.failure\_count >= self.failure\_threshold:

self.state = "OPEN"

print(f"Circuit breaker OPEN - {self.failure\_count} failures")

raise e

class ErrorRecoveryManager:

"""Gerenciador de recuperação de erros"""

def \_\_init\_\_(self):

self.circuit\_breakers: Dict[str, CircuitBreaker] = {}

self.retry\_policies: Dict[str, Callable] = {

"exponential\_backoff": RetryPolicy.exponential\_backoff,

"fixed\_interval": RetryPolicy.fixed\_interval,

"linear\_backoff": RetryPolicy.linear\_backoff

}

def get\_circuit\_breaker(self, service\_name: str) -> CircuitBreaker:

"""Obtém circuit breaker para serviço"""

if service\_name not in self.circuit\_breakers:

self.circuit\_breakers[service\_name] = CircuitBreaker()

return self.circuit\_breakers[service\_name]

async def execute\_with\_recovery(self, func: Callable, service\_name: str,

retry\_policy: str = "exponential\_backoff",

use\_circuit\_breaker: bool = True) -> Any:

"""Executa função com estratégias de recuperação"""

if use\_circuit\_breaker:

circuit\_breaker = self.get\_circuit\_breaker(service\_name)

async def protected\_func():

return await circuit\_breaker.call(func)

else:

protected\_func = func

retry\_func = self.retry\_policies.get(retry\_policy, RetryPolicy.exponential\_backoff)

return await retry\_func(protected\_func)

**Implementação no Runtime**

class FlowScriptRuntimeWithErrorHandling(FlowScriptRuntime):

"""Runtime com tratamento avançado de erros"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.error\_recovery = ErrorRecoveryManager()

self.error\_handlers: Dict[str, Callable] = {}

self.global\_error\_handler: Optional[Callable] = None

def register\_error\_handler(self, error\_pattern: str, handler: Callable):

"""Registra handler específico para tipo de erro"""

self.error\_handlers[error\_pattern] = handler

def set\_global\_error\_handler(self, handler: Callable):

"""Define handler global para erros não tratados"""

self.global\_error\_handler = handler

async def handle\_error(self, error: Exception, context: Dict[str, Any]) -> Any:

"""Processa erro baseado nas regras definidas"""

error\_type = type(error).\_\_name\_\_

service\_name = context.get("service\_name", "unknown")

# Procura handler específico

for pattern, handler in self.error\_handlers.items():

if self.\_matches\_error\_pattern(error, pattern):

print(f"Handling error with pattern: {pattern}")

return await handler(error, context)

# Handler global

if self.global\_error\_handler:

return await self.global\_error\_handler(error, context)

# Fallback padrão

print(f"Unhandled error: {error}")

raise error

def \_matches\_error\_pattern(self, error: Exception, pattern: str) -> bool:

"""Verifica se erro corresponde ao padrão"""

error\_type = type(error).\_\_name\_\_

error\_message = str(error)

# Padrões específicos

if pattern == "\*":

return True

elif "." in pattern:

service, error\_name = pattern.split(".", 1)

return error\_name in error\_message or error\_name == error\_type

else:

return pattern in error\_message or pattern == error\_type

async def execute\_with\_error\_handling(self, func: Callable, context: Dict[str, Any]) -> Any:

"""Executa função com tratamento de erro completo"""

try:

return await func()

except Exception as e:

return await self.handle\_error(e, context)

# Exemplo de handlers de erro

async def handle\_timeout\_error(error: Exception, context: Dict[str, Any]) -> Any:

"""Handler para erros de timeout"""

service\_name = context.get("service\_name", "unknown")

print(f"Handling timeout for service: {service\_name}")

# Implementar estratégia específica para timeout

if service\_name == "CreditBureau":

# Usar serviço backup

backup\_service = context.get("backup\_service")

if backup\_service:

return await backup\_service.getScore(context.get("ssn"))

# Fallback genérico

raise Exception("Service temporarily unavailable")

async def handle\_rate\_limit\_error(error: Exception, context: Dict[str, Any]) -> Any:

"""Handler para rate limiting"""

print("Rate limit detected, implementing backoff strategy")

# Aguardar antes de retry

await asyncio.sleep(60) # 1 minuto

# Retry da operação original

original\_func = context.get("original\_function")

if original\_func:

return await original\_func()

raise error

# Configuração de error handling

runtime = FlowScriptRuntimeWithErrorHandling()

runtime.register\_error\_handler("timeout", handle\_timeout\_error)

runtime.register\_error\_handler("rate\_limit", handle\_rate\_limit\_error)

**Monitoramento e Debugging**

**Sistema de Métricas**

from dataclasses import dataclass

from typing import Dict, List, Any

import time

import json

@dataclass

class Metric:

"""Representação de uma métrica"""

name: str

value: Any

timestamp: float

tags: Dict[str, str]

class MetricsCollector:

"""Coletor de métricas para workflows"""

def \_\_init\_\_(self):

self.metrics: List[Metric] = []

self.counters: Dict[str, int] = {}

self.gauges: Dict[str, float] = {}

self.histograms: Dict[str, List[float]] = {}

def increment\_counter(self, name: str, tags: Dict[str, str] = None):

"""Incrementa contador"""

self.counters[name] = self.counters.get(name, 0) + 1

self.\_record\_metric(name, self.counters[name], tags or {})

def set\_gauge(self, name: str, value: float, tags: Dict[str, str] = None):

"""Define valor de gauge"""

self.gauges[name] = value

self.\_record\_metric(name, value, tags or {})

def record\_histogram(self, name: str, value: float, tags: Dict[str, str] = None):

"""Registra valor em histograma"""

if name not in self.histograms:

self.histograms[name] = []

self.histograms[name].append(value)

self.\_record\_metric(name, value, tags or {})

def \_record\_metric(self, name: str, value: Any, tags: Dict[str, str]):

"""Registra métrica com timestamp"""

metric = Metric(

name=name,

value=value,

timestamp=time.time(),

tags=tags

)

self.metrics.append(metric)

def get\_summary(self) -> Dict[str, Any]:

"""Retorna resumo das métricas"""

summary = {

"counters": self.counters.copy(),

"gauges": self.gauges.copy(),

"histograms": {}

}

# Calcular estatísticas dos histogramas

for name, values in self.histograms.items():

if values:

summary["histograms"][name] = {

"count": len(values),

"min": min(values),

"max": max(values),

"avg": sum(values) / len(values),

"p50": self.\_percentile(values, 50),

"p95": self.\_percentile(values, 95),

"p99": self.\_percentile(values, 99)

}

return summary

def \_percentile(self, values: List[float], percentile: int) -> float:

"""Calcula percentil"""

sorted\_values = sorted(values)

index = int(len(sorted\_values) \* percentile / 100)

return sorted\_values[min(index, len(sorted\_values) - 1)]

class WorkflowTracer:

"""Sistema de tracing para workflows"""

def \_\_init\_\_(self):

self.traces: Dict[str, Dict] = {}

self.active\_spans: Dict[str, str] = {}

def start\_trace(self, trace\_id: str, workflow\_name: str) -> None:

"""Inicia novo trace"""

self.traces[trace\_id] = {

"workflow\_name": workflow\_name,

"start\_time": time.time(),

"spans": [],

"status": "running"

}

def start\_span(self, trace\_id: str, span\_name: str, parent\_span: str = None) -> str:

"""Inicia novo span dentro de um trace"""

span\_id = f"{span\_name}\_{int(time.time() \* 1000)}"

span = {

"span\_id": span\_id,

"name": span\_name,

"start\_time": time.time(),

"parent\_span": parent\_span,

"tags": {},

"logs": []

}

if trace\_id in self.traces:

self.traces[trace\_id]["spans"].append(span)

self.active\_spans[span\_id] = trace\_id

return span\_id

def finish\_span(self, span\_id: str, status: str = "success") -> None:

"""Finaliza span"""

if span\_id in self.active\_spans:

trace\_id = self.active\_spans[span\_id]

trace = self.traces[trace\_id]

for span in trace["spans"]:

if span["span\_id"] == span\_id:

span["end\_time"] = time.time()

span["duration"] = span["end\_time"] - span["start\_time"]

span["status"] = status

break

del self.active\_spans[span\_id]

def add\_span\_tag(self, span\_id: str, key: str, value: str) -> None:

"""Adiciona tag ao span"""

if span\_id in self.active\_spans:

trace\_id = self.active\_spans[span\_id]

trace = self.traces[trace\_id]

for span in trace["spans"]:

if span["span\_id"] == span\_id:

span["tags"][key] = value

break

def add\_span\_log(self, span\_id: str, message: str, level: str = "info") -> None:

"""Adiciona log ao span"""

if span\_id in self.active\_spans:

trace\_id = self.active\_spans[span\_id]

trace = self.traces[trace\_id]

for span in trace["spans"]:

if span["span\_id"] == span\_id:

span["logs"].append({

"timestamp": time.time(),

"message": message,

"level": level

})

break

def finish\_trace(self, trace\_id: str, status: str = "success") -> None:

"""Finaliza trace"""

if trace\_id in self.traces:

trace = self.traces[trace\_id]

trace["end\_time"] = time.time()

trace["duration"] = trace["end\_time"] - trace["start\_time"]

trace["status"] = status

def get\_trace(self, trace\_id: str) -> Dict:

"""Retorna dados do trace"""

return self.traces.get(trace\_id, {})

class FlowScriptDebugger:

"""Debugger avançado para FlowScript"""

def \_\_init\_\_(self, runtime):

self.runtime = runtime

self.breakpoints: Dict[str, List[str]] = {} # workflow\_name -> [steps]

self.watchpoints: Dict[str, List[str]] = {} # workflow\_name -> [variables]

self.step\_mode = False

self.debug\_sessions: Dict[str, Dict] = {}

def set\_breakpoint(self, workflow\_name: str, step\_name: str) -> None:

"""Define breakpoint em step específico"""

if workflow\_name not in self.breakpoints:

self.breakpoints[workflow\_name] = []

self.breakpoints[workflow\_name].append(step\_name)

print(f"Breakpoint set: {workflow\_name}.{step\_name}")

def set\_watchpoint(self, workflow\_name: str, variable\_name: str) -> None:

"""Define watchpoint em variável"""

if workflow\_name not in self.watchpoints:

self.watchpoints[workflow\_name] = []

self.watchpoints[workflow\_name].append(variable\_name)

print(f"Watchpoint set: {workflow\_name}.{variable\_name}")

def start\_debug\_session(self, instance\_id: str) -> None:

"""Inicia sessão de debug"""

self.debug\_sessions[instance\_id] = {

"started": time.time(),

"step\_history": [],

"variable\_history": [],

"paused": False

}

async def check\_breakpoint(self, instance\_id: str, workflow\_name: str, step\_name: str) -> bool:

"""Verifica se deve pausar em breakpoint"""

should\_pause = False

# Verifica breakpoints

if (workflow\_name in self.breakpoints and

step\_name in self.breakpoints[workflow\_name]):

shoul