
UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

GABRIEL TARQUINIO SALES MUNIZ

# SISTEMA DE COLETA DE DADOS BASEADO NA IEC 61499

Recife
2025

GABRIEL TARQUINIO SALES MUNIZ

# SISTEMA DE COLETA DE DADOS BASEADO NA IEC 61499

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Orientador(a): Prof. Dr. Herbert de Albérico de Sá Leitão

Recife
2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Muniz, Gabriel Tarquinio Sales.
Sistema de coleta de dados baseado na IEC 61499 / Gabriel Tarquinio Sales Muniz. - Recife, 2025.
100 p. : il., tab.

Orientador(a): Herbert Albérico De Sá Leitão
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia de Controle e Automação - Bacharelado, 2025.
Inclui referências, apêndices.

1. IEC 61499. 2. Coleta de Dados. 3. Indústria 4.0. 4. Sistemas Distribuídos. 5. Banco de dados não-relacionais. I. Leitão, Herbert Albérico De Sá. (Orientação). II. Título.

620 CDD (22.ed.)

GABRIEL TARQUINIO SALES MUNIZ

SISTEMA DE COLETA DE DADOS BASEADO NA IEC 61499

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Aprovado em: 17/12/2025

**BANCA EXAMINADORA**

_____

Prof. Dr. Herbert de Albérico de Sá Leitão (Orientador)
Universidade Federal de Pernambuco

_____

Prof. Dr. Jeydson Lopes da Silva
Universidade Federal de Pernambuco

_____

Prof. MSc. Néstor Iván Medina Giraldo
Universidade Federal de Pernambuco

# AGRADECIMENTOS

Primeiramente, gostaria de agradecer a minha mãe, Meiriangela Sales, por ter acreditado em mim desde do começo e por ter me dado suporte na minha vida e em todas as minhas escolhas.

Agradeço ao meu pai, Iverton José, por ter sido sempre um modelo para mim e por me ensinar que os estudos e o trabalho são apenas uma parte da vida.

Agradeço a minha irmã, Giovanna Tarquinio, por sempre ter sido a minha parceira e por ter me dado suporte em diferentes âmbitos da minha vida.

Agradeço a Peruquinha, meu cachorro e coautor não oficial deste TCC, por estar sempre no meu pé e por me proporcionar momentos de distração e lazer em uma rotina agitada.

Agradeço a Luana Karolyna, amor da minha vida, por me mostra que existe beleza neste mundo fora de cálculos, códigos e maquinas, além do incentivo a leitura e fazer o *cappucino* mais gostoso para me ajudar nos estudos.

Agradeço a João Veras, cunhado e amigo, por ter me emprestado a ESP32 que foi usada neste trabalho.

Agradeço ao resto da minha família, por estar sempre comigo em momentos importantes, pelas comemorações nas conquistas e pelos os consolos nas derrotas.

Agradeço aos meus amigos, em especial João Ferreira e Claudio José, por sempre estarem comigo nos momentos de pausa e de lazer.

Agradeço a equipe de robótica Maracatronics, por ter me proporcionado um ambiente de muito aprendizado e por ter me dado oportunidade de trabalhar com diferentes tipos de pessoas.

Agradeço as pessoas da Qualihouse Automação, por terem me proporcionado a experiencia de trabalhar no mercado de trabalho e pelo os ensinamentos técnicos e humanos para lidar com esse ambiente.

Agradeço ao professor Herbert, por ter me orientado no desenvolvimento deste trabalho e pelo o ensino do conteúdo da disciplina de Automacão e Controle de Sistemas Distribuídos, que foi essencial para o desenvolvimento deste projeto.

Agradeço aos meus colegas de curso, por estarem ao meu lado ao enfrentar os desafios da graduação.

Agradeço aos outros professores, pelo os ensinamentos passados durante toda a graduação.

E por fim agradeço a mim mesmo por ter sido resiliente e por ter acredito até o fim que daria certo.

“Insanidade é fazer a mesma coisa repetidamente e esperar resultados diferentes.” Albert Einstein.

## RESUMO

A crescente digitalização da indústria, impulsionada pelos princípios da Indústria 4.0, tem intensificado a necessidade de sistemas eficientes de coleta, armazenamento e análise de dados. Nesse contexto, este trabalho apresenta o desenvolvimento de um sistema de coleta de dados fundamentado na norma IEC 61499, que se destaca por sua abordagem orientada a sistemas distribuídos. A proposta visa integrar Controladores Lógicos Programáveis (CLPs) e softCLPs por meio do protocolo OPC UA, garantindo comunicação padronizada e interoperabilidade. O sistema é composto por duas etapas principais: a coleta de dados em dispositivos de campo e o armazenamento em um banco de dados não relacional, implementado através de blocos de interface de serviço. A adoção da IEC 61499 permite maior flexibilidade, escalabilidade e modularidade na arquitetura, favorecendo a digitalização dos processos industriais e servindo como base para a aplicação de tecnologias emergentes, como Internet Industrial das Coisas (IIoT), Big Data e computação em nuvem. Os resultados esperados incluem maior visibilidade operacional, suporte à manutenção preventiva e embasamento para tomadas de decisão estratégicas.



**Palavras-chave:** IEC 61499; Coleta de Dados, Indústria 4.0, Sistemas Distribuídos, Banco de dados não-relacionais.

## ABSTRACT

The increasing digitalization of the industry, driven by the principles of Industry 4.0, has intensified the need for efficient systems for data collection, storage, and analysis. In this context, this work presents the development of a data collection system based on the IEC 61499 standard, which stands out for its distributed systems-oriented approach. The proposal aims to integrate Programmable Logic Controllers (PLCs) and softPLCs through the OPC UA protocol, ensuring standardized communication and interoperability. The system consists of two main stages: data collection from field devices and storage in a non-relational database, implemented through service interface blocks. The adoption of IEC 61499 allows for greater flexibility, scalability, and modularity in the architecture, favoring the digitalization of industrial processes and serving as a foundation for the application of emerging technologies, such as the Industrial Internet of Things (IIoT), Big Data, and cloud computing. The expected outcomes include greater operational visibility, support for preventive maintenance, and a basis for strategic decision-making.



**Keywords**: IEC 61499; Data Collection, Industry 4.0, Distributed Systems, Non-relational Databases.

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CLP | Controlador Logico Programável |
| FB | Function Blocks |
| IEC | International Electrotechnical Commission |
| IoT | Internet of Things |
| IIoT | Industrial Internet of Things |
| TSDB | banco de dados de séries temporais |
| CSV | valores separados por vírgulas |
| SoftCLP | Controlador Logico Programável implementado em software |
| ECC | gráfico de controle de execução |
| SGBD | Sistema de Gerenciamento de Banco de Dados |
| GPIO | Entrada/Saída de propósito geral |
| SIFB | Service Interface Function Blocks |
| OPC UA | Arquitetura Unificada de Comunicações de Plataforma Aberta |

# SUMÁRIO

# 1 INTRODUÇÃO

A indústria 4.0 tem promovido mudanças significativas nos sistemas de produção, introduzindo tecnologias digitais, como Internet of Things(IoT), ou internet das coisas em português, e *Big Data*, e promovendo a integração de sistemas físicos à redes inteligentes. Nesse contexto, os dados assumem o papel central, uma vez que fornecem informações essenciais sobre o estado dos processos produtivos, possibilitando sua otimização contínua, a implementação de estratégias de manutenção preditiva e a tomada de decisões autônomas, suportadas por sistemas de inteligência artificial (ARNALSON; BREMDAL; SOLVANG, 2022).

Dessa forma, intensificou-se a demanda por sistemas de controle distribuídos, capazes de integrar e coordenar estruturas produtivas cada vez mais complexas. A partir dessa necessidade, surge a norma IEC 61499 (MERKUMIANS; GSELLMANN; SCHITTER, 2021), que determina e padroniza a implementação de sistemas de controle distribuídos. As principais vantagens da norma IEC 61499 incluem o uso de uma linguagem de fácil compreensão baseada em diagramas de blocos de funções. Seu diferencial está na extensão do conceito tradicional de blocos de funções, permitindo sua adaptação à realidade dos sistemas de controle distribuído. Além disso, a norma foi concebida para atender a três requisitos fundamentais: portabilidade, configurabilidade e interoperabilidade (PANG *et al*., 2014).

Um outro efeito associado ao conceito de Indústria 4.0 é a crescente digitalização das indústrias, aumentando a geração de dados vindo de sensores e variáveis do sistema. Com isso, também cresce a necessidade de criar meios mais eficientes para coleta e análise desses dados gerados (KAJOLA, 2024). Um exemplo é o estudo de Paavo Kajola (2024), que propõe um sistema de coleta e análise de dados multivariados em séries temporais, baseado na norma IEC 61499. No estudo, a aquisição de dados teve como finalidade capturar dados de sistemas industriais modernos, coletar dados de servidores OPC UA, armazenar os dados em um banco de dados de séries temporais (TSDB) e preparar os dados para análise distribuída utilizando blocos de interface de serviços.

A implementação de sistemas de coleta de dados na indústria apresenta inúmeras vantagens, uma vez que proporciona maior visibilidade dos processos produtivos, reduz desperdícios de recursos e possibilita a identificação de gargalos operacionais. No contexto de sistemas distribuídos, o tráfego de grandes volumes de informações entre CLPs torna a presença de um sistema de coleta indispensável para aprimorar o desempenho, viabilizar a manutenção preventiva e apoiar a tomada de decisões estratégicas (KAJOLA, 2024).

No escopo da Indústria 4.0, a utilização e o processamento de dados ocorrem, de forma geral, em quatro etapas principais: (i) Coleta de dados, na qual informações são extraídas de sensores, motores e sistemas de controle; (ii) Armazenamento, etapa em que os dados coletados são registrados em bancos de dados ou arquivos CSV; (iii) Pré-processamento, fase responsável pela filtragem e eliminação de dados inválidos ou redundantes; e (iv) Análise, em que gráficos e algoritmos de inteligência artificial são aplicados para extrair padrões e gerar *insights* relevantes (BASANTA-VAL, 2017).

Nesse cenário, a adoção da norma IEC 61499 para o desenvolvimento de sistemas de coleta de dados revela-se especialmente vantajosa. Como os dados industriais são provenientes de diferentes processos e dispositivos, o sistema de coleta caracteriza-se, em essência, como um sistema distribuído, alinhando-se diretamente à filosofia da IEC 61499. Além disso, a coleta de dados representa o primeiro passo para a digitalização industrial, constituindo-se como base para a implementação de tecnologias emergentes, tais como a Internet Industrial das Coisas (IIoT), *Big Data* e computação em nuvem.

Inspirado por essa abordagem, o presente trabalho tem como objetivo projetar e implementar um sistema de coleta e armazenamento de dados em conformidade com a norma IEC 61499. Para tanto, foi desenvolvida uma arquitetura em que a etapa de coleta é realizada por um CLP, enquanto o armazenamento é executado por um softCLP em operação em um PC central. A comunicação entre os dispositivos ocorre por meio da rede, utilizando o protocolo OPC UA. No processo de armazenamento, foram implementados blocos de interface de serviço responsáveis por registrar os dados em um banco de dados não relacional.

## 1.1 Objetivos

### *1.1.1 Geral*

Desenvolver um sistema de coleta e armazenamento de dados industriais fundamentado na norma **IEC 61499**, visando aprimorar a visibilidade dos processos, facilitar a digitalização da indústria e fornecer suporte à análise de dados para tomada de decisões estratégicas.

### *1.1.2 Específicos*

- **Projetar** uma arquitetura distribuída de coleta de dados baseada em CLPs e softCLPs conforme a norma IEC 61499;
- **Implementar** a comunicação entre os dispositivos utilizando o protocolo **OPC UA**, garantindo interoperabilidade e padronização;
- **Desenvolver** blocos de interface de serviço para o armazenamento de dados em um banco de dados não relacional acessado localmente pelo *softCLP*;
- **Validar** o sistema por meio de simulações e testes que demonstrem sua eficiência na coleta e no registro de informações industriais.

## 1.2 Organização do Trabalho

Este trabalho está estruturado da seguinte forma: No Capítulo 2, apresenta-se a fundamentação teórica, contemplando os principais conceitos relacionados à norma IEC 61499, suas ferramentas e aspectos pertinentes a bancos de dados, de modo a oferecer a base conceitual necessária para a compreensão do estudo.

O Capítulo 3 descreve a metodologia adotada, detalhando as etapas que orientaram o desenvolvimento do trabalho, desde o planejamento até a execução. No capítulo 4, é mostrado como foi feito o desenvolvimento do trabalho. No capítulo 5 é mostrado os resultados obtidos neste projeto. Por fim, o Capítulo 6 reúne as conclusões obtidas a partir dos resultados apresentados no capítulo anterior, bem como sugestões e perspectivas para trabalhos futuros.

# 2 FUNDAMENTAÇÃO TEÓRICA

## 2.1 Norma IEC 61499

A norma IEC 61499 foi desenvolvida com o objetivo de oferecer um ambiente prático e flexível para a implementação de sistemas de controle distribuídos. Em contraste com a norma IEC 61131-3, que se concentra na padronização das linguagens de programação para CLPs tradicionais e está vinculada a arquiteturas centralizadas e dependentes de hardware específico (INTERNATIONAL ELECTROTECHNICAL COMMISSION, 2025), a IEC 61499 introduz um modelo orientado a sistemas distribuídos, com foco na interoperabilidade e na independência de plataforma (PANG *et al*., 2014).

Esse novo paradigma facilita a comunicação entre dispositivos de diferentes fabricantes e garante a portabilidade das aplicações em softwares distintos (CHRISTENSEN *et al.*, 2012). Por exemplo, uma solução desenvolvida para um CLP pode ser facilmente adaptada e executada em plataformas alternativas, como o Raspberry Pi, sem a necessidade de reescrita completa do projeto.

Além da portabilidade, a IEC 61499 foi concebida para atender a dois requisitos adicionais essenciais:

- **Configurabilidade**: possibilita a alteração dinâmica dos parâmetros de um projeto sem a necessidade de interromper sua execução, permitindo maior flexibilidade e adaptabilidade dos sistemas.
- **Interoperabilidade**: assegura que dois ou mais dispositivos, mesmo de fabricantes distintos, possam executar de forma coordenada todas as aplicações de um sistema distribuído.

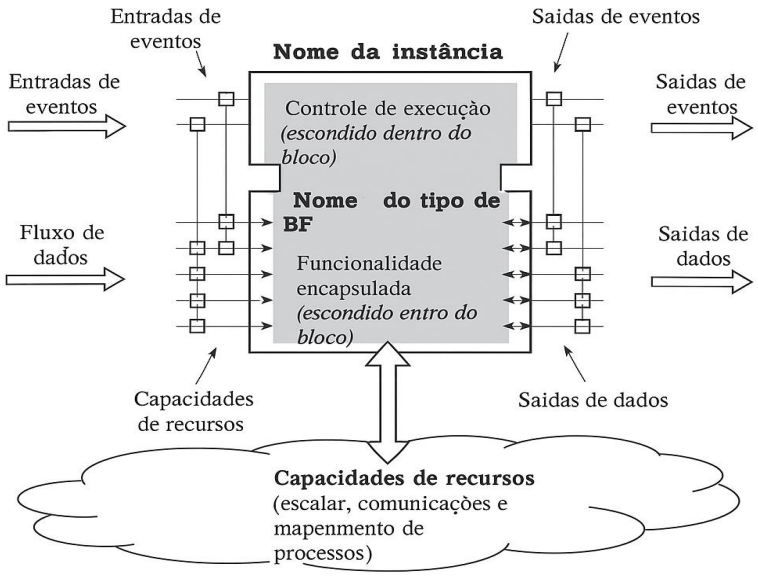
Dessa forma, a IEC 61499 representa uma evolução significativa em relação à IEC 61131-3, ao alinhar-se às demandas da Indústria 4.0 e às necessidades de arquiteturas de automação mais distribuídas, escaláveis e independentes de hardware (CHRISTENSEN *et al.*, 2012).

O elemento central da norma IEC 61499 que possibilita essas funcionalidades é o conceito de blocos de função. Esses blocos representam abstrações de código que permitem a incorporação de múltiplos algoritmos, podendo ser desenvolvidos tanto em linguagens definidas pela IEC 61131-3, como o texto estruturado, quanto em linguagens de alto nível, como C++ e Python (PINTO, 2014).

Outra característica fundamental dos blocos definidos pela IEC 61499 é que eles são orientados a eventos, em contraste com os blocos da IEC 61131-3, que seguem um modelo baseado em ciclos de varredura. Essa abordagem orientada a eventos contribui para sistemas mais eficientes e confiáveis no tratamento de eventos, reduzindo atrasos desnecessários e otimizando a utilização dos recursos computacionais (LEWIS; ZOTIL, 2014).

Sua representação gráfica é constituída por 2 partes: cabeçalho e o corpo. No cabeçalho são definidos os eventos de entrada, os eventos de saída e o ECC (*execution control chart, gráfico de controle de execução*), onde é administrado a ocorrência dos eventos e no corpo se encontram os dados de entrada, os algoritmos encapsulados e os dados de saída (LEWIS; ZOTIL, 2014).

Figura 1: Representação do bloco de funções da norma IEC 61499




Fonte: adaptado do LEWIS (2014)

## 2.2 Tipos de Blocos

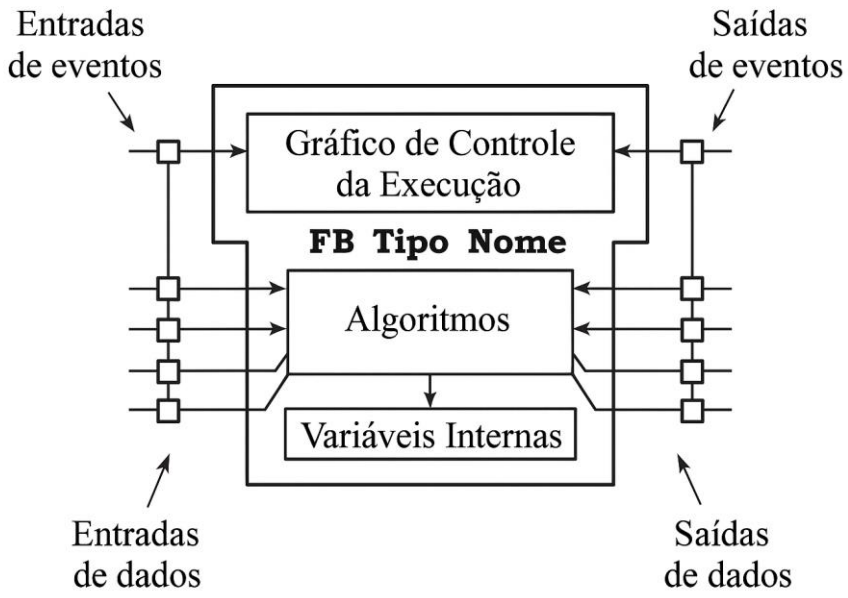
Existem 3 tipos de blocos que são usados na IEC 61499 que são:

- **Basic Function Blocks:** São blocos que tem um ou mais algoritmos encapsulados neles e sua execução é baseada nos eventos de entrada.
- **Composite Function Blocks:** São blocos que tem um conjunto de outros blocos encapsulados neles.
- **Service Interface Function Blocks:** Os blocos de interface de serviço são blocos que permitem o acesso externo do sistema, sendo útil para ler sensores, usar atuadores e fazer comunicação em rede.

### *2.2.1 Basic Function Blocks*

O tipo de bloco básico é composto por um ou vários algoritmos que são executados a partir da ocorrência dos eventos de entrada e são gerenciados pelo ECC.

Figura 2: Bloco Básico de funções




Fonte: adaptado do LEWIS (2014)

O **ECC** é um componente fundamental em aplicações desenvolvidas com base na norma IEC 61499, pois é responsável por gerenciar as transições de estados de um bloco de funções. A cada evento recebido, o ECC avalia as condições de transição e executa o algoritmo associado ao estado ativo, processando os dados de entrada e gerando as saídas correspondentes. Essa abordagem orientada a eventos permite maior controle sobre o fluxo de execução, aumentando a previsibilidade e a modularidade do sistema (LIAKH *et al.*, 2022).

A estrutura do ECC é composta por três elementos principais:

- **Estados**: representam diferentes modos de operação de um bloco de funções, cada um associado a ações ou algoritmos específicos.

- **Transições**: definem as condições sob as quais o ECC altera o estado atual, geralmente disparadas por eventos.

- **Ações**: correspondem aos algoritmos executados quando um estado é ativado, permitindo o processamento dos dados e a geração de respostas apropriadas.

A seguir, apresenta-se um exemplo de ECC aplicado a um bloco básico de funções, ilustrando um programa simples de somador.

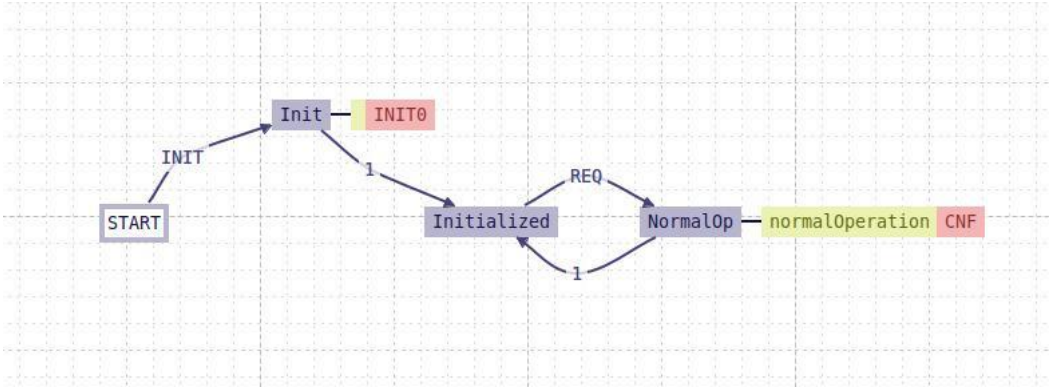
O ECC (Execution Control Chart) desse bloco somador é composto por quatro estados: START, que representa o estado inicial do bloco; Init; Initialized; e NormalOp.

O ciclo de funcionamento do ECC inicia-se no estado START. Quando o bloco recebe o evento INIT, ele transita para o estado Init, gerando o evento de saída INIT0 e, em seguida, passando para o estado Initialized. Nesse ponto, o programa conclui a fase de configuração (*setup*) e entra na fase de execução cíclica (*loop*).

Ao receber o evento REQ, o bloco transita para o estado NormalOp, no qual é executado o algoritmo normalOperation. Esse algoritmo realiza a soma dos dois dados de entrada. Após a execução, é gerado o evento de saída CNF, e o bloco retorna ao

estado Initialized, aguardando uma nova ocorrência do evento REQ, que reinicia o ciclo descrito.

Figura 3: ECC de um bloco básico

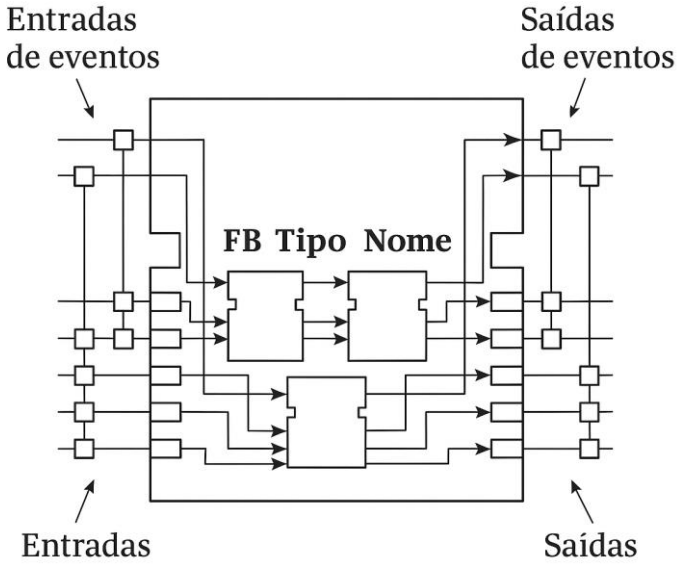



Fonte: Do próprio autor

### *2.2.2 Composite Function Blocks*

Para definir o comportamento de blocos compostos, utiliza-se uma rede de blocos internos, o ECC e algoritmos, como ocorre nos blocos básicos. A Figura 4 apresenta a estrutura de um bloco composto.

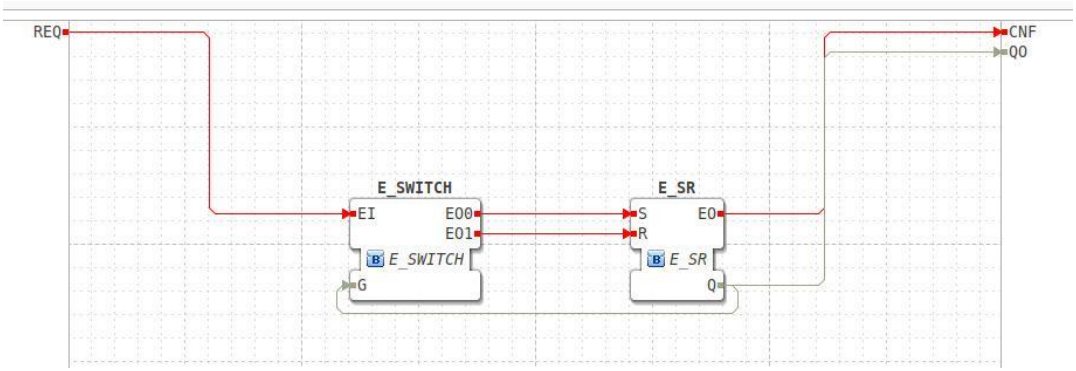
Figura 4: Representação de um bloco composto




Fonte: Adaptado do LEWIS (2014)

Abaixo temos um exemplo de um bloco composto que tem encapsulado nele dois blocos em sequência que são o E_Switch e o E_SR que são usados em conjuntos para criar uma aplicação de *flip flop,* que são circuitos sequenciais em que o valor de saída depende do valor atual do componente.

Figura 5: ECC de um bloco composto



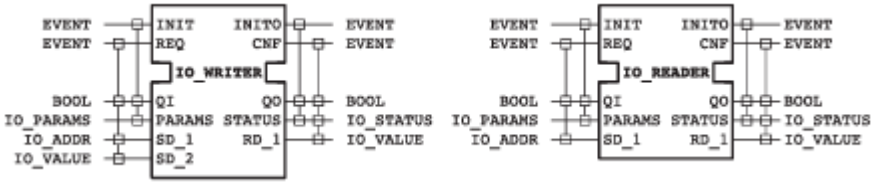

Fonte: Do próprio autor

### *2.2.3 Blocos de interface de serviços*

Os blocos de interface de serviço (SIFB – Service Interface Function Blocks) são componentes utilizados para estabelecer a comunicação entre a aplicação e o ambiente externo. Esse ambiente externo pode corresponder tanto a dispositivos de *hardware*, permitindo a leitura de sinais de entrada e a escrita em sinais de saída, quanto a redes de comunicação entre diferentes dispositivos, viabilizando a integração necessária para a implementação de sistemas distribuídos.

Por exemplo, os blocos PUBLISHER e SUBSCRIBER, representados na figura 6, possibilitam a comunicação em rede utilizando diversos protocolos, como *sockets*, OPC UA, Modbus TCP, HTTP, entre outros. Já os blocos IO_Read e IO_Write permitem, respectivamente, a leitura de valores provenientes de sensores físicos conectados ao hardware e o envio de comandos para atuadores.

Figura 6: Representação de blocos SIFBs

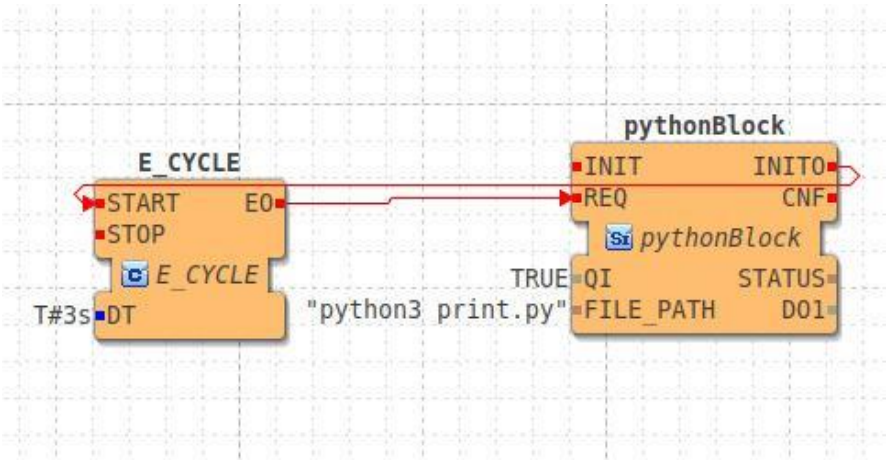



Fonte: LEWIS (2014)

Outra aplicação para os blocos de interface de serviços é acessar arquivos que estão armazenados em um hardware em que o projeto está sendo executado, como um arquivo txt ou CSV e códigos que foram escritos usando linguagens de alto nível.

Nesse projeto, será mostrado como usar esse tipo de bloco para executar códigos em Python que são externos ao projeto, são chamados a partir do código dos blocos que são gerados em C++. Na figura 7 temos um bloco chamado PythonBlock que foi criado para auxiliar esse projeto, a função deste bloco é executar códigos feitos em Python, bastando apontar o diretório do arquivo em FILE_PATH. O código está disponível no Apêndice.

Figura 7: Exemplo de uso de um bloco de interface




Fonte: Do próprio autor

## 2.3 4Diac

O 4Diac é um kit de desenvolvimento de código aberto composto por um ambiente de desenvolvimento integrado (IDE) e a máquina FORTE, que executa os FBs da IEC 61499, utilizado para o desenvolvimento, implementação e visualização de aplicações industriais de sistemas de controle distribuídos com base na norma IEC 61499.

Embora existam outras ferramentas compatíveis com a norma IEC 61499, nas próximas seções deste trabalho é apresentada uma análise comparativa entre o FDBK e o 4Diac, a fim de identificar qual delas oferece maior eficiência e adequação para o desenvolvimento de aplicações voltadas a sistemas de controle distribuído.

No estudo de (PANG *et al*., 2014), foi realizada uma investigação sobre portabilidade e semântica entre diferentes ferramentas baseadas na IEC 61499. Além do 4Diac, foram avaliadas outras três plataformas: FBDK, IsaGRAF e nxtStudio, considerando a capacidade de portabilidade entre elas.

Tabela 1: Portabilidade entre as principais ferramentas baseadas na norma IEC 61499

| | FBDK | 4Diac | nxtStudio | IsaGRAF |
|---|---|---|---|---|
| FBDK | - | Full | Partial | N/A |
| 4Diac | Full | - | Partial | N/A |
| nxtStudio | Partial | Partial | - | N/A |
| IsaGRAF | N/A | N/A | N/A | - |

Fonte: adaptado (PANG *et al*., 2014).

Com os resultados apresentados na Tabela 1, nota-se que as soluções nxtStudio e isaGRAF não se mostram adequadas para utilização, devido à sua baixa compatibilidade com outras ferramentas e à escassez de informações disponíveis.

### *2.3.1 4Diac vs FDBK*

O FBDK foi a primeira ferramenta produzida com o objetivo de demonstrar os conceitos da norma IEC 61499, seus componentes são um editor de texto e um

ambiente de execução (FBRT) que utiliza um modelo de execução baseado em múltiplas *threads*, onde cada *thread* é uma unidade de execução de tarefas, não preemptivo com escalonamento em profundidade, suporta a sintaxe XML normativa, o que facilita a troca de elementos com outras ferramentas compatíveis e sua limitação é o seu desenvolvimento na linguagem Java que limita seu suporte a hardware industrial(PANG *et al*., 2014).

O 4Diac é uma ferramenta de código aberto que foi desenvolvida originalmente pela academia para estudar e desenvolver projetos baseados na norma IEC 61499, seus componentes são um editor baseado no Eclipse e um ambiente de execução chamado FORTE que foi construído usando C++, seu modo de execução é um modelo sequencial com despache de eventos, altamente compatível com FDBK pois ambos seguem a sintaxe XML como define a norma e tem como mérito ser a única ferramenta ainda ativa de código aberto que recebe atualizações e é a ferramenta mais utilizada para pesquisa e desenvolvimento(PANG *et al*., 2014).

Com isso foi definido que a melhor ferramenta para utilizar nesse projeto é o 4Diac pois além dos itens apresentados acima, ele possui uma interface agradável e intuitiva para desenvolvimento.

### *2.3.2 Componentes do 4Diac*

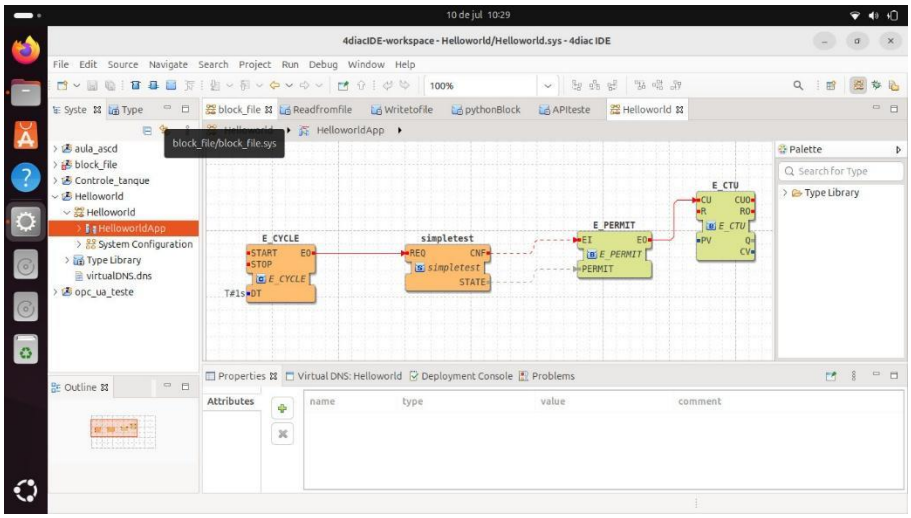
O 4Diac tem dois componentes principais para a execução de controle de sistemas distribuídos sob a norma IEC 61499.

- **4Diac FORTE**: O FORTE é um ambiente de tempo de execução de múltiplas tarefas baseado na norma IEC 61499, que oferece suporte à execução de redes de blocos de função em dispositivos embarcados de pequeno porte. Trata-se de uma implementação compacta, de baixo custo e baixo consumo de memória, desenvolvida em C++. Além disso, o FORTE já foi testado em diversos sistemas operacionais, incluindo Windows, Linux e FreeRTOS(ECLIPSE FOUNDATION, 2025).

**4Diac IDE:** É um ambiente escrito em Java baseado na IDE do Eclipse, a IDE do 4Diac fornece ao usuário várias ferramentas para desenvolver aplicações de

sistemas distribuídos baseados na norma IEC 61499 como criar bloco de funções, criar aplicativos, configurar dispositivos e realizar outras tarefas relacionadas a norma IEC 61499 (ECLIPSE FOUNDATION, 2025). O 4DIac IDE também vem com várias bibliotecas nativas pré-compiladas e prontas para serem executadas na máquina FORTE, a coleção de bibliotecas do 4Diac é vasta e bastante rica, tem blocos da norma IEC 61131-3 que podem ser usados em aplicações de sistemas de controle distribuídos, tem a biblioteca Convert onde ficam blocos destinados a conversão de dados, tem a biblioteca Net que hospeda os blocos que realizam a comunicação em rede e várias outras.

Figura 8: 4Diac IDE




Fonte: Do próprio autor

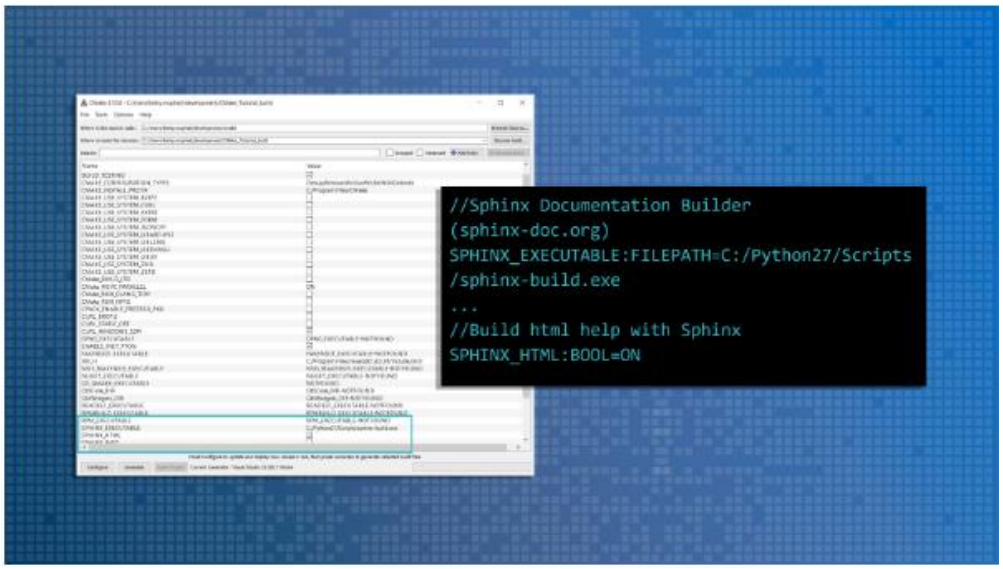
## 2.4 Cmake

Outro *software* essencial no desenvolvimento de aplicações para sistemas de controle distribuído é o CMake. Trata-se de um conjunto de ferramentas de código aberto (*open-source*) que permite criar, testar e empacotar aplicações de forma automatizada e multiplataforma (KITWARE, 2025).

No contexto deste trabalho, o CMake é utilizado como ferramenta de compilação e geração da máquina FORTE. A partir dos arquivos disponibilizados no pacote 4Diac FORTE, é possível gerar máquinas FORTE personalizadas, capazes de executar tanto blocos de função desenvolvidos pelo próprio usuário quanto blocos nativos presentes nas bibliotecas fornecidas pelo 4Diac.

O CMake pode ser utilizado de duas formas: por meio da linha de comando (*terminal*) ou por meio de uma interface gráfica, conforme ilustrado na figura 9:

Figura 9: Meios para usar o Cmake




(KITWARE, 2025)

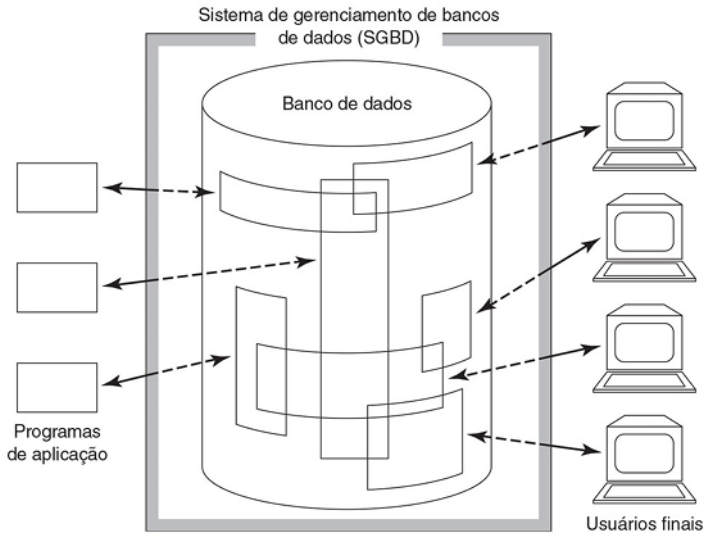
## 2.5 Banco de Dados

O banco de dados pode ser definido como um sistema computacional destinado ao armazenamento e à manutenção de informações, composto por hardware, software e pelos próprios dados. As informações armazenadas em um banco de dados possuem caráter persistente, sendo removidas apenas mediante uma solicitação explícita, ao contrário dos dados de entrada e saída dos blocos de função da IEC 61499 que são atualizados de forma constante (DATE, 1991).

O que gerencia um banco de dados é o SGBD (Sistema de Gerenciamento de Banco de Dados), que é responsável por intermediar o acesso ao banco de dados atuando entre as camadas do usuário e o armazenamento físico, oferecendo uma interface lógica e intuitiva para manipulação de dados.

As principais funções do SGBD são gerenciar como os dados são armazenados e acessados, permitir operações para manipulação de dados, estabelecer um controle de acesso, garantindo múltiplos acessos de usuários de forma simultânea e sem conflitos, definir permissões sobre quais usuários podem acessar o banco de dados, proteger os dados contra acessos não autorizados e garantir a integridade e a independência das informações (DATE, 1991).

Figura 10:Representação do Banco de dados




Fonte: (DATE, 1991)

A linguagem padrão usada para manipulações de dados é o SQL, que significa Structured Query Language (linguagem de consulta estruturada), com ela é possível criar tabelas, inserir dados, consultar dados, alterar dados, excluir dados e tabelas. Os principais comandos usados para realizar essas ações estão listados abaixo:

- CREATE TABLE: usado para criação de tabelas;
- DROP TABLE: usado para remover tabelas;
- INSERT: usado para inserir dados em uma tabela;

- SELECT: usado para realizar consulta de dados;
- UPDATE: usado para alterar dados inseridos;
- DELETE: usado para excluir dados;

### *2.5.1 Bancos Relacionais*

Os bancos de dados relacionais são sistemas que organizam as informações por meio de regras estruturais bem definidas. Do ponto de vista estrutural, esses bancos são compostos por tabelas, nas quais os dados são armazenados e relacionados entre si. Todas as tabelas que compõem um banco de dados são compostas por linhas e colunas, cada coluna já tem definido o tipo de valor que ela vai comportar – por exemplo: inteiro, real, texto e data – e em cada tabela é necessário que uma coluna seja a chave primária para identificação de uma linha específica registrada na tabela. Os vínculos criados entre as tabelas são referenciados através de chaves estrangeiras, onde a coluna que representa uma chave primária em uma tabela aparece em outra tabela como chave estrangeira (DATE, 1991).

Esse tipo de banco de dados foi proposto por Edgar F. Codd em 1970 e ainda é bastante utilizado em vários tipos de sistemas modernos porque o seu aspecto estrutural é bastante intuitivo e fácil de entender, sendo ideal para uma organização maior no armazenamento de dados. Outra vantagem que torna o banco de dados relacional atrativo é a sua integridade em relação aos dados, estabelecendo certas restrições em relação ao armazenamento de dados em estados não válidos e respeitando a relação das tabelas, garantindo que as chaves primárias não sejam nulas e que as relações das chaves estrangeiras estejam devidamente validadas (DATE, 1991).

Porém, para aplicações industriais, haverá uma alta velocidade de geração de dados descentralizados e desconexos o que torna a escolha dos bancos de dados relacionais não apropriada para armazenamento, pois além do baixo desempenho para lidar com um alto volume de dados, é exigida uma relação entre os dados distribuídos que não necessariamente vai existir. Por isso que o tipo de banco de dados não-relacional é mais utilizado para aplicações IIoT do que o banco de dados relacional.

### *2.5.2 Bancos Não-Relacionais*

Com a necessidade de manipular grande quantidade de dados que surgiram depois do advento da *web* 2.0, foram criados os bancos de dados não relacionais também conhecido como NoSQL (“not only SQL”) que tem como características um sistema flexível, os bancos de dados possuem código aberto, os dados não são estruturados e a escalabilidade de armazenamento costuma ser mais barata e menos complexa (GARCIA & SOTTO,2019).

Os bancos de dados não-relacionais são divididos nos seguintes grupos:

- Orientado a documentos: São coleções de atributos e valores que podem assumir tipos de valores diferentes, essas coleções são armazenadas no formato JSON. Os tipos de bancos mais populares são o MongoDB e o CouchDB (DIANA & GEROSA, 2010).
- Armazéns chave-valor: São armazenados objetos indexados por chaves exclusivas que possibilitam fazer a busca por esses objetos. Um exemplo de banco que está nessa categoria é o Amazon DynamoDB (AMAZON, 2025).
- Banco de dados por Grafos: Diferente dos outros tipos de banco de dados que não definem um modelo de dados previamente, aqui é definido como modelo de grafos. Onde os dados são representados por estruturas de grafos que são constituídas por nós e bordas. Exemplos de banco de dados: Property Graph e o Resource Description Framework (RDF) (AMAZON, 2025).

Após analisar essas e várias outras categorias, foi decidido utilizar um banco de dados orientado a documentos por causa do funcionamento simples, da arquitetura bastante flexível e por ter bancos de dados gratuitos como o MongoDB, que foi o banco de dados escolhido para armazenar os dados lidos pelo o sistema de coletas.

#### *2.5.2.1 MongoDB*

O MongoDB, como mencionado no tópico anterior, é um banco de dados orientado a documentos que permite armazenamento de coleções de documentos

semelhantes ao JSON. Ele foi criado com o objetivo de ser uma alternativa a bancos relacionais como o MySQL para suprir a necessidade de sistemas modernos que têm um volume grande de dados. Além de gratuito, o MongoDB possui uma comunidade ampla e ativa, bem como documentação abrangente sobre instalação, uso de APIs e desenvolvimento de aplicações, disponíveis para consulta (MONGODB Inc, 2025).

Além dessas características, cada documento é um objeto independente que está organizado no esquema chave-valor, podendo ter diferentes tipos de dados como arrays, texto e números reais. A vantagem dessa estrutura é que ela elimina a necessidade de definir um esquema fixo, permitindo que documentos de uma mesma coleção tenham estruturas diferentes. Essa característica é um dos pontos principais que torna o MongoDB uma excelente escolha para lidar com aplicações de Big Data e Internet Industrial das Coisas (IIoT).

No MongoDB também é possível fazer uma replicação dos dados em diferentes servidores, assim garantindo a disponibilidade e recuperação desses dados em caso de falha. Nesse banco de dados também é permitido particionar os dados em vários servidores diferentes, possibilitando o aumento de desempenho, da capacidade de armazenamento e tornando possível a escalabilidade horizontal dos dados.

# 3 METODOLOGIA

A metodologia proposta neste trabalho visa a criação, validação e aplicação de um sistema de coleta de dados, conforme a norma IEC 61499, voltados à manipulação de dados industriais em ambientes compatíveis com os princípios da Indústria 4.0. A abordagem está dividida em cinco etapas principais: (1) definição dos requisitos, (2) modelagem funcional, (3) desenvolvimento dos blocos, (4) testes e validação, e (5) integração em um sistema piloto.

## 3.1 Definição dos Requisitos Funcionais e de Dados

Nesta etapa, foram definidos os requisitos do projeto, que consistem na criação de um sistema de coleta de dados baseado na norma IEC 61499, capaz de executar duas das quatro fases do tratamento de dados: a coleta e o armazenamento em um banco de dados. Como o sistema foi idealizado para coletar informações provenientes de ambientes industriais, ele deve ser capaz de se comunicar por meio de um protocolo industrial e registrar esses dados em um banco que suporte grandes volumes de informações, além de oferecer mecanismos adequados para lidar com possíveis perdas de dados.

Tendo em vista esses dois pré-requisitos, definiu-se que o protocolo de comunicação a ser utilizado será o OPC UA, em razão da sua facilidade de integração com diferentes tipos de sistemas. Conforme mencionado na Seção 2.5.2, o banco de dados escolhido para o armazenamento das informações é o MongoDB.

## 3.2 Modelagem Funcional com Blocos IEC 61499

Com base nos requisitos, o sistema de coleta de dados terá um bloco de rede que vai executar o protocolo OPC UA, um bloco que fará a comunicação com o banco de dados MongoDB e alguns blocos do tipo evento que serão necessários para gerar eventos de requisição e encerramento do banco de dados. Na biblioteca de blocos do

4Diac, existem blocos de comunicação em rede e blocos de eventos, logo é necessário apenas criar o bloco que fará a comunicação com o MongoDB.

## 3.3 Desenvolvimento e Implementação dos Blocos

O desenvolvimento da interface gráfica do bloco MongoDB foi feito na IDE do 4Diac. Nesta etapa, foi definido o número de eventos de entrada e de saída do bloco e a quantidade de dados de entrada e de saída. Posteriormente, o *script* interno do bloco MongoDB, apresentado no Apêndice A, foi desenvolvido em linguagem C++.

Para a realização do teste de robustez, foram criados mais dois blocos o bloco cronometroFB que obtém o tempo em que o bloco MongoDB leva para fazer um armazenamento no banco de dados e o bloco csv_block que cria um arquivo CSV que armazena o tempo de ocorrência da chegada dos dados e o tempo obtido do bloco cronometroFB. A interface gráfica dos dois blocos também foi desenvolvida no 4Diac IDE.

No desenvolvimento deste trabalho foram desenvolvidos dois blocos que não foram usados na versão final deste projeto, mas as suas funcionalidades são interessantes para fazer uso em futuros projetos que são o myBD e o pythonBlock. O myBD é um bloco de interface de serviço que faz comunicação com o banco de dados Sqlite3, um banco de dados relacional. O pythonBlock é um bloco de interface de serviço que executa códigos em python.

## 3.4 Testes e Validação

O sistema de coleta de dados foi validado em um ambiente de conexão local, no qual ocorreu a troca de informações entre o sistema desenvolvido e um *software* responsável pela leitura e transmissão de dados utilizando o protocolo OPC UA. Os testes consideraram:

- **Validação funcional**: verificar se os blocos executam corretamente as operações de manipulação de dados.

- **Robustez e confiabilidade**: verificação de perda de dados e avaliação de performance.

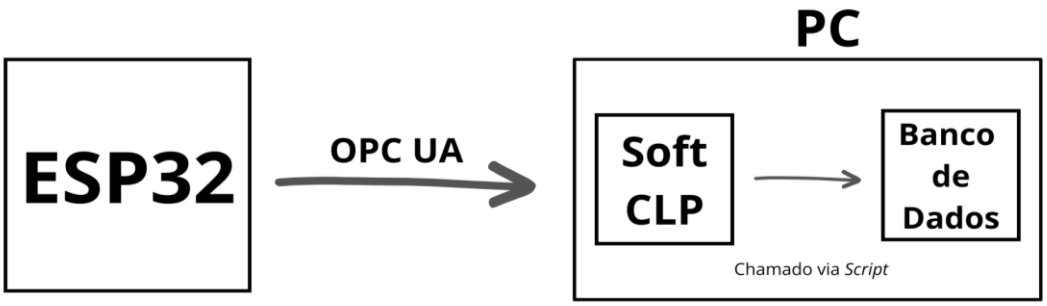
Ferramentas de monitoramento do 4diac foram utilizadas para observar o comportamento dos blocos em tempo real.

## 3.5 Integração em Sistema Piloto

Os blocos foram integrados em uma aplicação piloto, representando um cenário industrial simplificado. O sistema piloto foi um servidor OPC UA, operando em uma ESP32. Esse servidor foi utilizado para monitorar variáveis de processo como estados de *Relays* e a temperatura do ambiente.

A topologia de rede de uma planta industrial para aquisição de dados é formada por um CLP de campo que coleta os dados e o sistema que recebe os dados via rede, neste trabalho é proposta uma estrutura semelhante, mas que possui a vantagem de integrar em um mesmo dispositivo o sistema de controle (CLP) e o sistema de armazenamento de dados (banco de dados). A figura 11 apresenta esta topologia de rede, tendo a ESP32 como um dispositivo de campo.

Figura 11:Topologia de rede




Fonte: Do próprio autor

O softCLP se comunica com a ESP32 através do protocolo OPC UA, após o recebimento dos dados, o softCLP faz comunicação com o banco de dados via *script* (código de programação). Esse tipo de comunicação foi possível devido ao desenvolvimento do bloco SIFB MongoDB da IEC 61499, permitindo que o softCLP possa ter acesso direto a informações de outros serviços do PC.

Essa aplicação permite demonstrar a viabilidade prática da metodologia e os ganhos em termos de modularidade, interoperabilidade e uso de dados em tempo real.

# 4 DESENVOLVIMENTO DO TRABALHO

Neste capítulo, será apresentado o processo de desenvolvimento deste trabalho, abrangendo desde a configuração do ambiente de trabalho até a demonstração do seu funcionamento. O objetivo desta seção é detalhar os softwares utilizados, as etapas de configuração necessárias para sua operacionalização e os blocos de função desenvolvidos, juntamente com seus respectivos códigos.

Esta seção está organizada em três partes: **(1)** Configuração do ambiente de trabalho; **(2)** Desenvolvimento dos blocos de função; e **(3)** Ambiente de teste e integração com o sistema piloto.

## 4.1 Configuração do ambiente de trabalho

Neste trabalho, foi necessário fazer a instalação dos seguintes softwares para execução do sistema de coleta de dados:

- As ferramentas do 4Diac: Como foi dito na seção 2.3.2, o 4Diac tem duas ferramentas que são o 4Diac IDE e o 4Diac FORTE. As duas ferramentas foram necessárias para o desenvolvimento do projeto.

- Compiladores C/C++: Foram necessários para compilação e geração da máquina FORTE que executa o projeto do sistema de coleta de dados.

- Open62541: O pacote Open62541 da OPC Foundation é uma biblioteca de código aberto que habilita o protocolo OPC UA na máquina FORTE (OPEN62541, 2025).

- UaExpert: *Software* para comunicação OPC UA da *Unified Automation,* usado para validar o sistema de coleta de dados (UNIFIED AUTOMATION, 2025).

- Python: Foi necessário para fazer a comunicação com o banco de dados MongoDB.

- MongoDB: Banco de dados usado para fazer o armazenamento dos dados coletados (MONGODB, 2025).

Após a instalação de todas as ferramentas, foi necessário integrar a API do Python com a máquina FORTE usando os comandos da Figura 12 no arquivo cmakeList.txt principal da máquina FORTE.

Figura 12: Comandos usados para 4Diac FORTE reconhecer a API do Python

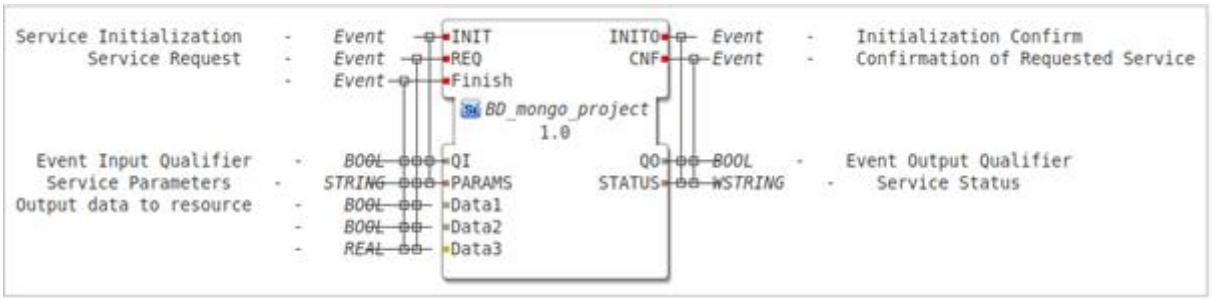

Fonte: Do próprio autor

## 4.2 Desenvolvimento do SIFB MongoDB

O bloco MongoDB foi desenvolvido com o propósito de estabelecer a comunicação entre a máquina FORTE, em execução no *softCLP*, e o banco de dados MongoDB instalado no sistema. Esse SIFB (Service Interface Function Block) também é responsável por realizar operações de criação do banco de dados e de suas coleções, bem como pela inserção de dados nesses repositórios.

O bloco foi projetado com três eventos de entrada, dois eventos de saída, cinco dados de entrada e dois de saída.

Figura 13:SIFB MongoDB




Fonte: Do próprio autor

O evento INIT, quando acionado, realiza a inicialização do bloco MongoDB. Durante esse processo, é executada uma verificação para assegurar que o bloco foi iniciado corretamente. Após essa verificação, o interpretador Python é inicializado. Ao término da execução do evento INIT, é gerado o evento de saída INIT0, indicando que o processo de inicialização foi concluído.

O evento REQ, ao ser acionado, seleciona o nome do banco de dados que será usado e a coleção em que esses dados serão inseridos a partir da entrada PARAMS, caso o banco de dados e a coleção não existam, elas são criadas. No evento REQ os dados que são recebidos nas entradas Data1, Data2 e Data3 são inseridos na coleção indicada. Após essa ação ser finalizada, o bloco aciona o evento de saída CNF.

O evento FINISH tem a função de encerrar a conexão com a API do Python. Essa ação é necessária devido à natureza da própria API, pois, caso seja inicializada e finalizada repetidamente em um curto intervalo de tempo, poderá ocorrer um erro na máquina FORTE, ocasionando a interrupção do sistema. Ao término da execução do evento FINISH, é gerado o evento CNF, indicando a conclusão do processo.

Os cinco dados de entrada são QI, PARAMS, Data1, Data2 e Data3. O QI é uma variável booleana usada para habilitar a inicialização do bloco, a variável PARAMS é uma *Wstring* que recebe os parâmetros de configuração para acesso ao banco de dados. Esses parâmetros são separados por ponto e vírgula: o primeiro corresponde ao nome do banco de dados, o segundo ao nome da coleção e os demais aos nomes dos atributos. Como três dados são armazenados, foram definidos três atributos. As variáveis Data1, Data2 e Data3 representam os dados que serão

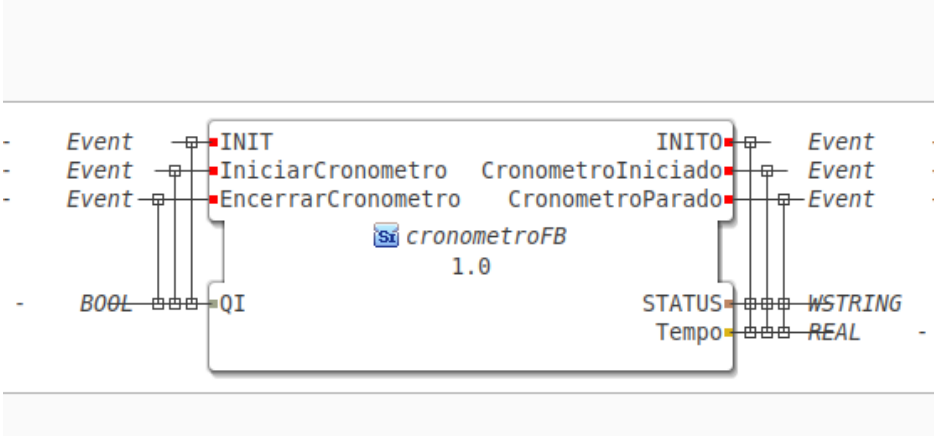
enviados ao MongoDB. Os valores Data1 e Data2 são variáveis booleanas que indicam o estado de dois relés, enquanto Data3 é uma variável do tipo Real, destinada a armazenar a leitura de um sensor de temperatura.

Os dois dados de saída são usados para monitoramento do bloco, sendo o Q0 uma variável booleana e o STATUS uma variável do tipo *Wstring*. O código desse bloco está disponível no Apêndice A.

## 4.3 Desenvolvimento do SIFB cronometroFB

O bloco cronometroFB foi criado com o propósito de obter o tempo em que o bloco MongoDB leva para fazer armazenamento de dados, a interface gráfica do bloco está representada na figura 14.

Figura 14:Bloco cronometroFB




Fonte: Do próprio autor

O bloco cronometroFB possui três eventos de entrada, três eventos de saída, um dado de entrada e dois de saída. O evento INIT ao ser acionado, executa as configurações iniciais do bloco e gera o evento de saída INIT0.

O evento IniciarCronometro ao ser acionado, começa a contar o tempo e gera o evento de saída CronometroIniciado. Após o acionamento do evento

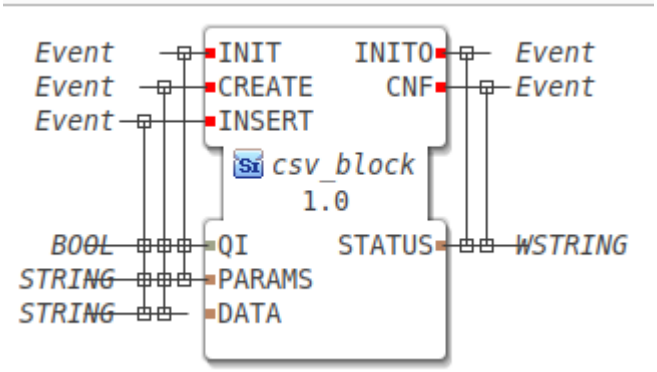
EncerrarCronometro o tempo para de ser contado e o evento de saída CronometroParado é acionado.

Este bloco possui o dado de entrada booleana QI que é usado para habilitar a inicialização do bloco, o dado de saída STATUS, que é uma variavel do tipo Wstring, monitora o estado em que o bloco se encontra. O bloco cronometroFB também retorna o dado de saída tempo, nele está contido o tempo contado pelo o bloco cronometroFB. Seu código está disponível no Apêndice B.

## 4.4 Desenvolvimento do SIFB csv_block

O bloco csv_block foi criado para salvar dados temporais gerados pelo o sistema de coleta de dados, representado pela figura 15, o bloco possui 3 eventos de entrada, 2 eventos de saída, 3 dados de entrada e 1 de saída.

Figura 15:Bloco csv_block




Fonte: Do próprio autor

O evento INIT inicializa o bloco e gera o evento de saída INIT0 e gera o evento INIT0, o evento CREATE cria o arquivo csv e gera o evento de saída CNF e o evento INSERT faz a inserção dos dados da entrada DATA no arquivo CSV e gera o evento de saída CNF.

O dado de entrada QI habilita a inicialização do bloco, o dado de entrada PARAMS está contido nos parâmetros de criação do arquivo que são o nome do

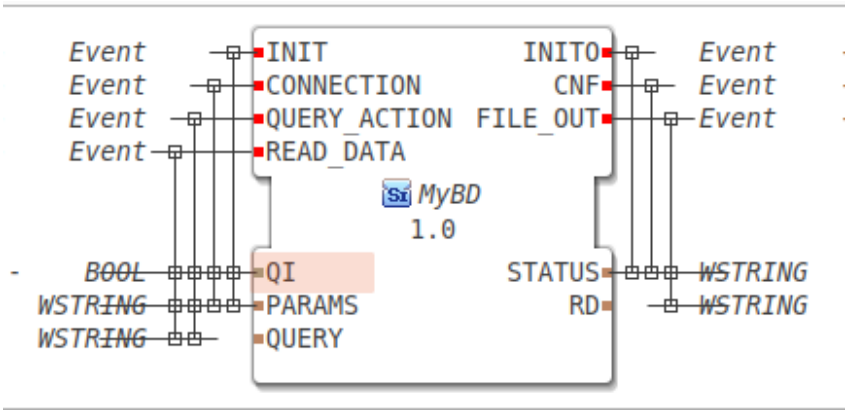
arquivo e os nomes das colunas, caso seja necessário. Os parâmetros são separados por vírgula.

O dado de entrada DATA contém os dados que são inseridos no arquivo csv, em caso de inserção de mais de um dado é necessário concatenar usando vírgula. o dado de saída STATUS, que é uma variavel do tipo Wstring, monitora o estado em que o bloco se encontra. Seu código está disponível no Apêndice D.

## 4.5 Desenvolvimento do SIFB myBD

O bloco myBD foi criado para fazer conexão com o banco de dados Sqlite3, a interface gráfica do bloco está representada na figura 16, mostrando que o bloco tem 4 eventos de entrada, 3 eventos de saída, 3 dados de entrada e 2 dados de saída.

Figura 16:Bloco de interface de serviço que faz comunicação com o SQlite3




Fonte: Do próprio autor

O evento INIT inicializa o bloco MyBD e gera o evento de saída INIT0, o evento CONNECTION estabelece a conexão com o banco de dados e gera o evento de saída CNF, o evento QUERY_ACTION executa o codigo sql na variável de entrada QUERY, gerando o evento de saída CNF e READ_DATA executa códigos de operações de consulta e retorna o resultado no dado de saída RD, gerando o evento de saída FILE_OUT.

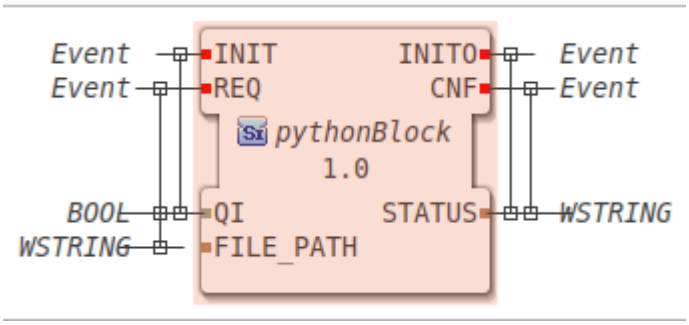
A entrada do dado de entrada booleana QI, é usado para habilitar a inicialização do bloco, o dado de saída STATUS, que é uma variável do tipo Wstring,

monitora o estado em que o bloco se encontra. A entrada PARAMS é definido o nome do banco de dados que o bloco fará conexão e o dado de entrada QUERY armazena o código sql que será utilizado. O código está disponível no Apêndice B.

## 4.6 Desenvolvimento do SIFB pythonBlock

O bloco pythonBlock foi criado para fazer a execução de arquivos .py, que são códigos escritos em python. A interface gráfica do bloco está representada na figura 17, onde o bloco tem 2 eventos de entrada, 2 eventos de saída, 2 dados de entrada e 1 de saída.

Figura 17:Bloco pythonBlock




Fonte: Do próprio Autor

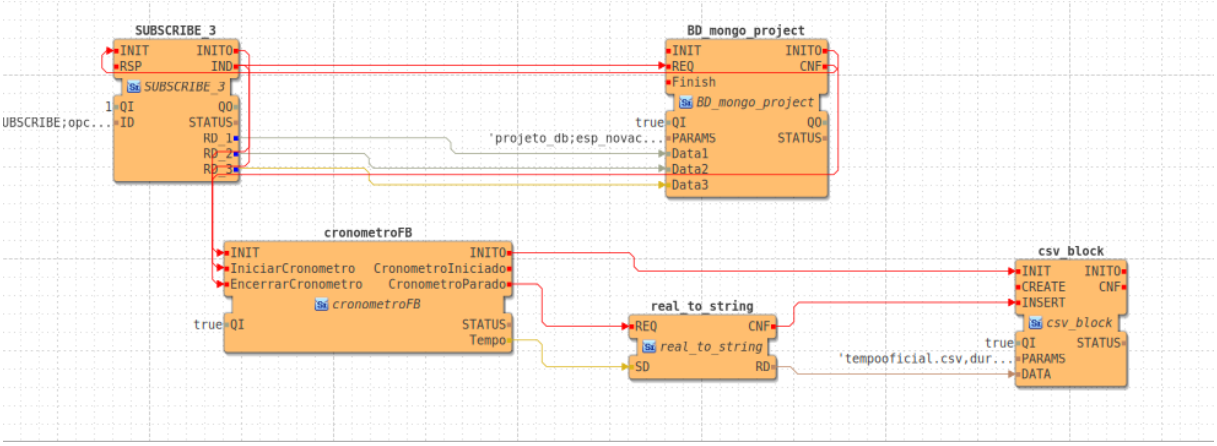
O evento INIT inicia o bloco pythonBlock e gera o evento de saída INIT0, o evento REQ faz uma requisição ao bloco para executar o arquivo para onde o diretório contido na variável FILE_PATH está apontando e gera o evento CNF. O código deste bloco encontra-se disponível no Apêndice E.

## 4.7 Desenvolvimento da aplicação 4Diac

O sistema de coleta de dados foi desenvolvido com o objetivo de obter informações provenientes de um controlador em campo, utilizando o protocolo OPC UA. Esse protocolo industrial é independente de plataforma de *software* e *hardware*, o que o torna particularmente adequado para aplicações baseadas na norma IEC 61499 (KAJOLA, 2024). A seguir, apresenta-se a aplicação do sistema de coleta

executada no soft-CLP. Na sequência, são descritos os blocos que compõem essa aplicação, bem como suas respectivas funcionalidades.

Figura 18: Aplicação 4Diac




Fonte: Do próprio autor

Como apresentado na Figura 18, além do bloco MongoDB, também são utilizados os blocos CronometroFB e csv_block, que, em conjunto, têm a função de contabilizar o intervalo de tempo necessário para que o bloco MongoDB armazene uma informação e salve em um arquivo. No arquivo CSV também ficou registrado o tempo em que ocorreram os registros dos dados no arquivo, isso foi usado para calcular os intervalos em que ocorreram a coleta de dados. Essas informações são utilizadas para a avaliação da performance e da robustez do sistema. Os códigos dos blocos CronometroFB e csv_block estão disponíveis no APÊNDICE.

Também é empregado o bloco SUBSCRIBE_3, localizado na pasta *Net* do 4Diac, responsável pela comunicação com a ESP32 por meio do protocolo OPC UA.

Para configurar esse bloco, é necessário, primeiramente, habilitar a variável QI com o valor true e, em seguida, definir o identificador ID. A estrutura desse identificador segue o formato:

opc_ua[<action>;<endpoint>;<pair1>;<pair2>;<pair3>

Em que *action*, ou ação em português, representa o tipo de ação que o bloco executará seja de leitura ou escrita, *endpoint*, ou ponto final em português,

corresponde ao endereço do servidor ao qual o bloco será conectado, e os demais parâmetros (pair1, pair2, pair3) indicam os dados com os quais o bloco irá interagir.

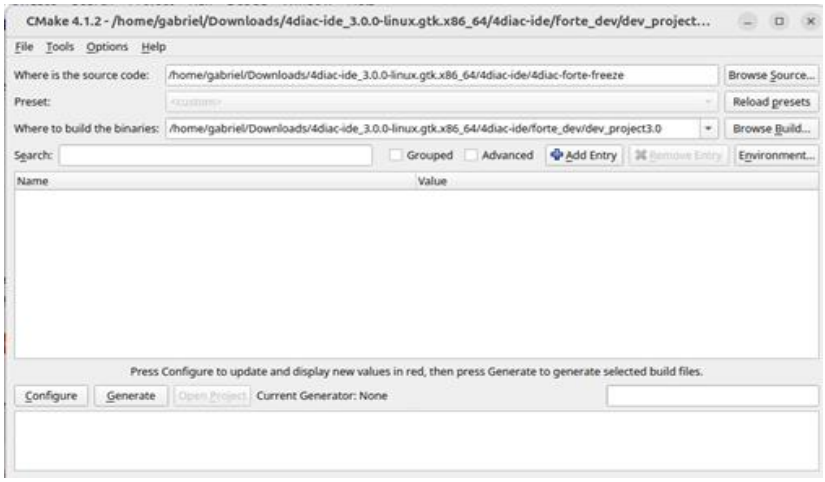
Inicialmente, o sistema de coleta de dados foi validado por meio de uma conexão local com um *software* capaz de realizar comunicação via OPC UA. Assim, a configuração inicial do parâmetro ID foi definida como:

opc_ua[READ;/Objects/1:Control Relay number 0;/Objects/1:Control Relay number 1;/Objects/1:Ambient temperature]

## 4.8 Geração da máquina FORTE

Nesta etapa, a geração da máquina FORTE do projeto é realizada utilizando o CMake. Como mencionado na Seção 2.4, existem duas opções para o uso do CMake: a interface gráfica e o terminal. Optou-se pela interface gráfica por ser mais intuitiva, como ilustrado na Figura 19, além de facilitar a visualização dos parâmetros configurados.

Figura 19: Interface Gráfica do Cmake




Fonte: Do próprio autor

Os parâmetros utilizados para configuração da máquina FORTE que necessitam de alteração são:

- CMAKE_BUILD_TYPE: Serve para definir o tipo de compilação que o Cmake irá fazer, normalmente é escolhido a opção Debug;
- FORTE_ARCHITECTURE: Neste parâmetro foi definido o tipo de sistema em que o *runtime* foi gerado e executado, em um ambiente linux é escolhido Posix;
- FORTE_COM_OPCUA: Este parâmetro habilita a comunicação OPC UA no FORTE;
- FORTE_EXTERNAL_MODULES_DIRECTORY: Neste atributo, foi adicionado o diretório dos blocos desenvolvidos para este projeto;
- FORTE_MODULE_CONVERT: Habilita o uso dos blocos da pasta Convert;
- FORTE_MODULE_IEC61131: Habilita o uso dos blocos da pasta IEC61131;
- FORTE_MODULE_UTILS: Habilita o uso dos blocos da pasta Utils.

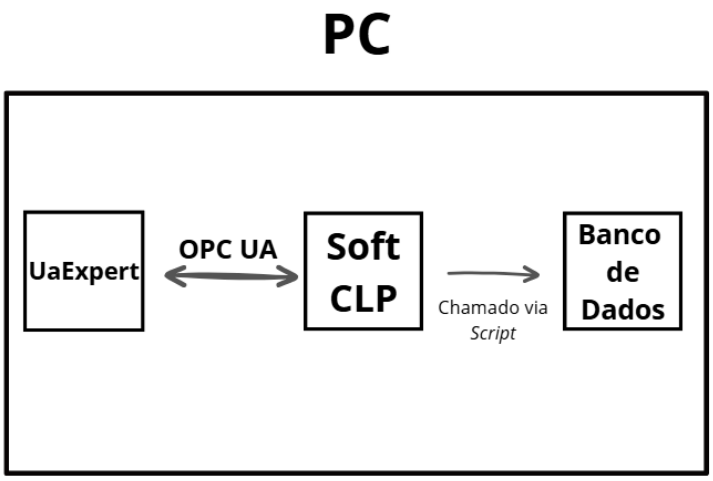
Após a configuração e geração dos arquivos, a máquina FORTE do projeto foi gerada.

## 4.9 Validação do Sistema de Coleta de Dados

Com a geração do runtime, a etapa seguinte consiste em testar o sistema de coleta de dados. Para isso, utilizou-se o software UaExpert, da Unified Automation, empregado para comunicação OPC UA e capaz de atuar tanto como cliente quanto como servidor OPC.

Nesse teste, a conexão entre o software e o softCLP é realizada localmente, conforme representado na Figura 16. Nessa configuração, o softCLP coleta os dados disponibilizados pelo UaExpert via OPC UA e, em seguida, armazena essas informações no banco de dados por meio de um script.

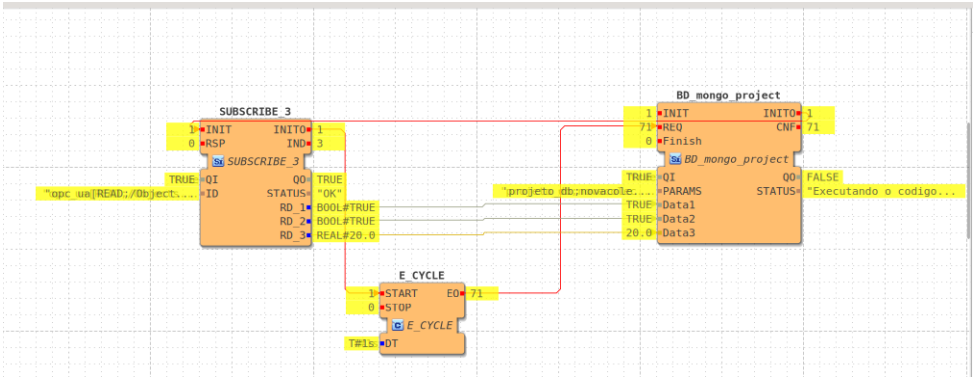
Figura 20: Topologia da conexão local




Fonte: Do próprio autor

Uma vez estabelecida a conexão local, foi possível visualizar a chegada dos dados coletados e o percurso realizado até o seu armazenamento no banco de dados. A execução do sistema de coleta de dados está representada na Figura 21.

Figura 21:Executando comunicação com UaExpert



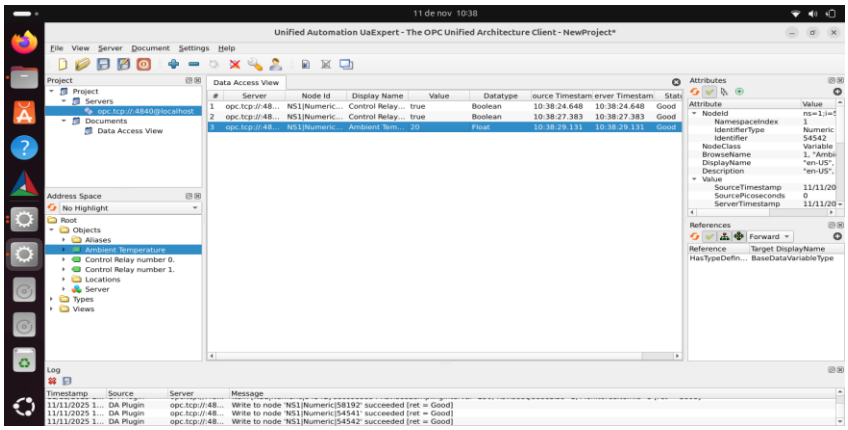

Fonte: Do próprio autor

Para realizar esse teste, foi necessário adicionar o bloco E_CYCLE, responsável por gerar as requisições destinadas ao bloco MongoDB. Isso ocorreu porque, durante a comunicação com o software UaExpert, o evento de saída IND do

bloco SUBSCRIBE_3 não é acionado, já que as variações nos dados são realizadas manualmente pelo usuário. Consequentemente, os blocos CronometroFB e CSV_BLOCK foram removidos, pois, nesta etapa, não houve avaliação do desempenho do sistema de coleta de dados.

Na Figura 22 está sendo mostrado a interface do UaExpert, nele é possível fazer a leitura e escrita de dados. Ao alterar os valores das variáveis, percebe-se a mesma alteração no sistema que está sendo executado no 4Diac, comprovando o funcionamento da comunicação entre o UaExpert e softCLP.

Figura 22: Leitura e escrita dos dados no UaExpert




Fonte: Do próprio autor

Após verificar a comunicação, o passo seguinte foi confirmar se o MongoDB estava armazenando os dados corretamente. Para isso, realizou-se uma consulta no banco de dados utilizado para o armazenamento, cujo resultado foi a exibição dos registros apresentados na Figura 23. Esse resultado comprova que o sistema de coleta de dados está funcionando adequadamente

Figura 23: Visualização dos dados armazenados no MongoDB


Fonte: Do próprio autor

## 4.10 Integração com o Sistema Piloto

O Sistema Piloto, conforme descrito anteriormente na Seção 3.5, consiste em um servidor OPC UA executado em uma placa ESP32. Esse servidor é responsável por monitorar os pinos GPIO32 e GPIO33, utilizados como *outputs* para ativar *Relays*, além do GPIO4, ao qual está conectado um sensor de temperatura. Os valores lidos são enviados a outros sistemas por meio do protocolo OPC UA para os sistemas que o acessam. A ESP32 usada neste trabalho foi a ESP32 DevkitV1, representada na Figura 24.

Figura 24:ESP32 DevkitV1


Fonte: (CIRCUITSTATE, 2023)

A ferramenta usada para configurar e programar a ESP32 foi a ESP-IDF, devido ao projeto que foi usado para executar o servidor OPC UA que está disponível no repositório Github em https://github.com/cmbahadir/opcua-esp32.git.

Com a inicialização do servidor OPC, o processo foi acompanhado através do terminal para verificar se o servidor está conectado à rede. Quando o terminal mostra a mensagem “Got a IP Event”, como é mostrado na Figura 25, significa que o servidor está conectado à rede.

Figura 25: Execução do servidor OPC

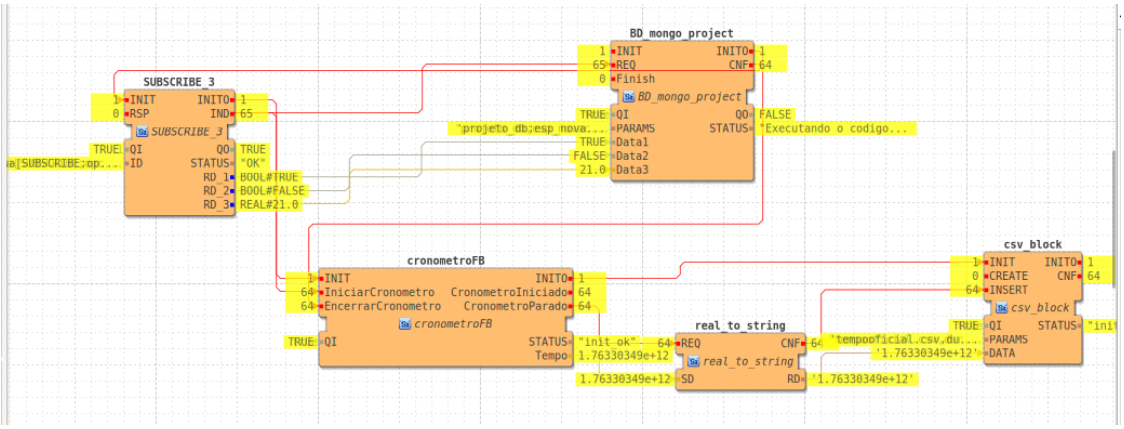
```
I (2069) wifi:dp: 1, bi: 102400, li: 3, scale listen interval from 307200 us to
307200 us
I (2119) wifi:<ba-add>idx:0 (ifx:0, 58:d9:d5:48:1c:41), tid:6, ssn:2, winSize:64
I (2349) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (5199) SNTP: Time is not set yet. Settting up network connection and getting t
ime over NTP.
I (5199) SNTP: Initializing SNTP
I (5199) SNTP: Waiting for system time to be set... (1/10)
I (7199) SNTP: Waiting for system time to be set... (2/10)
I (9199) SNTP: Waiting for system time to be set... (3/10)
I (9879) wifi:<ba-add>idx:1 (ifx:0, 58:d9:d5:48:1c:41), tid:0, ssn:0, winSize:64
I (9959) SNTP: Notification of a time synchronization event
I (11199) SNTP: Current time: 2025-11-10 22:17:16
I (11199) MEMORY: Heap size after OPC UA Task : 198476
I (11199) OPCUA_ESP32: Fire up OPC UA Server.
I (11199) esp_netif_handlers: sta ip: 192.168.81.100, mask: 255.255.255.0, gw: 1
92.168.81.1
I (11209) online_connection: Got IP event!
I (11209) online_connection: Connected to VIAWEB_IVERTON_33382360_p2
I (11219) online_connection: IPv4 address: 192.168.81.100
I (11219) main_task: Returned from app_main()
I (11469) OPCUA_ESP32: Heap Left : 43660
```

Fonte: Do próprio autor

Com a execução do servidor OPC UA iniciada, a próxima etapa foi testar a conexão dele com o sistema de coleta de dados. Para a conexão acontecer, foi necessário fazer mudanças no bloco SUBSCRIBE_3 em relação ao parâmetro ID, primeiramente o tipo de conexão muda pois a conexão deixa de ser local e passa a ser remota, logo o *endpoint* deixa de ser *localhost* e passa a ser o IP da ESP32. Devido a mudança do tipo de conexão, também foi necessário mudar o tipo de ação porque o bloco SUBSCRIBE não suporta fazer a operação READ quando a conexão é remota, logo a ação foi substituída por SUBSCRIBE.

Após a realização dessas alterações, a comunicação ocorreu adequadamente, sem apresentar falhas. Durante a execução no 4Diac mostrada na figura 26, observou-se uma performance maior do sistema em relação ao cenário da seção 4.5 porque o volume de dados foi maior. Além disso, ao verificar o armazenamento de dados no MongoDB, constatou-se que o processo foi realizado com sucesso, como foi demonstrado na Figura 27, comprovando a validade do sistema de coleta de dados na obtenção de informações provenientes de dispositivos remotos.

Figura 26:Sistema conectado a ESP32




Fonte: Do próprio autor

Figura 27:Dados armazenados da comunicação com a ESP32


Fonte: Do próprio autor

# 5 RESULTADOS

Este capítulo apresenta os resultados obtidos a partir da implementação, integração e validação do sistema de coleta de dados baseado na norma IEC 61499, bem como os testes de desempenho, robustez e confiabilidade realizados. Os resultados são organizados em quatro partes: (1) Funcionamento do sistema em ambiente local; (2) Integração com o sistema piloto baseado em ESP32; (3) Armazenamento de dados no banco MongoDB; e (4) Desempenho, robustez e confiabilidade do sistema.

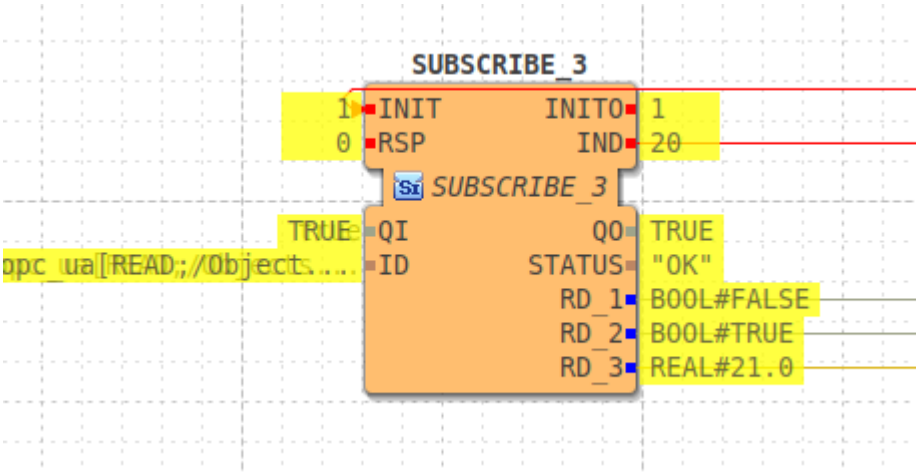
## 5.1 Funcionamento do Sistema em Ambiente Local

A primeira etapa de validação consistiu em executar o sistema de coleta de dados em um ambiente local, utilizando o software UaExpert como servidor OPC UA. Nessa configuração, o soft-CLP executando a máquina FORTE foi capaz de estabelecer comunicação com o UaExpert, ler as variáveis disponibilizadas e encaminhá-las ao bloco MongoDB para posterior armazenamento.

Durante esse teste, observou-se que:

- O sistema respondeu adequadamente às alterações manuais realizadas no UaExpert, foram realizados 20 inputs para esse teste;
- As variações nos valores das variáveis foram corretamente identificadas pelo bloco SUBSCRIBE_3, o número de inputs realizados foi igual ao número de eventos IND gerados no bloco como pode ser visto na imagem 28;
- As informações transmitidas foram armazenadas sem inconsistências no banco de dados, como foi visto na Figura 23.

Figura 28:Contagem de 20 eventos, correspondente a quantidade de inputs




Fonte: Do próprio autor

## 5.2 Integração com o Sistema Piloto (ESP32)

A segunda etapa envolveu a integração com o sistema piloto, composto por uma ESP32 executando um servidor OPC UA responsável por monitorar dois relés (GPIO32 e GPIO33) e um sensor de temperatura (GPIO4).

Após as adaptações necessárias no bloco SUBSCRIBE_3 – incluindo a substituição do tipo de ação READ por SUBSCRIBE e a mudança do *endpoint* para o IP da ESP32 – a comunicação foi estabelecida tendo como principais resultados:

- O envio de dados contínuo pela ESP32, permitindo testar o sistema sob maior volume de informações;
- Um registro com menor latência por parte do softCLP, em comparação ao teste local;

Como resultado, os dados provenientes da ESP32 foram armazenados corretamente no MongoDB. Nenhuma falha de comunicação foi observada durante esta execução, confirmando a operação do sistema em cenário realista de coleta industrial com uma taxa de atualização (latência) de 100 ms. Essa etapa comprovou a capacidade do sistema de lidar com dados provenientes de dispositivos remotos, reforçando sua aplicabilidade em ambientes de Indústria 4.0.

## 5.3 Armazenamento de Dados no MongoDB

Em ambas as etapas de validação, o banco MongoDB apresentou comportamento consistente e adequado ao propósito do sistema. Nas consultas realizadas durante os testes, verificou-se que:

- Todos os documentos gerados foram armazenados com estrutura correta;
- Não foram observadas perdas ou duplicações irregulares de dados;
- O tempo de escrita dos documentos permaneceu estável, independentemente do volume de dados processados;

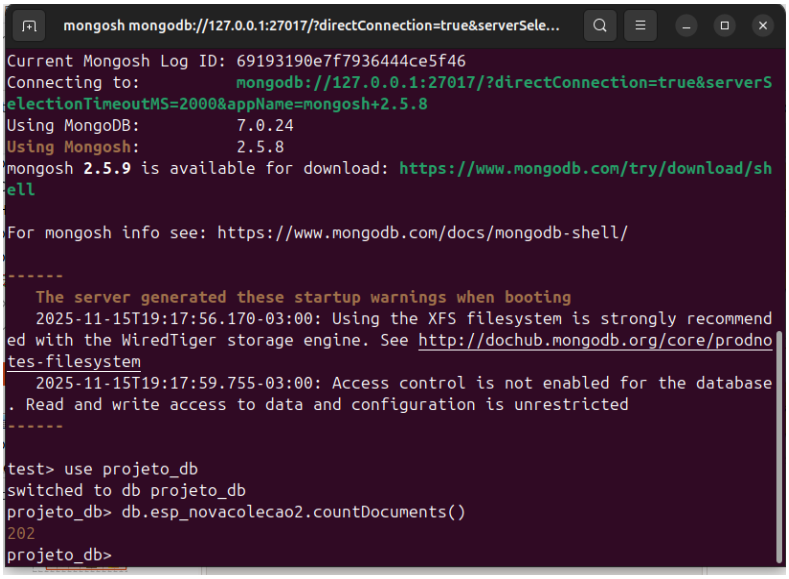
Os resultados confirmam que o MongoDB atendeu às necessidades de escalabilidade e flexibilidade que são exigidas por sistemas de coleta de dados industriais, especialmente por permitir armazenamento sem necessidade de esquema fixo.

## 5.4 Teste de Confiabilidade

O teste de confiabilidade consiste em executar o sistema de coleta de dados e monitorar o número de ocorrências do evento de requisição do bloco MongoDB. Após um número específico de ocorrências, o sistema é parado e o banco de dados é consultado para verificação do número de registros feito após aquela execução usando o comando db.nome_colecao.countDocuments(). Se o número de documentos for próximo do número de ocorrências do evento de requisição, significa que o sistema de coleta de dados é confiável.

Neste teste, o número específico escolhido para o número de ocorrências foi 200. O número de eventos de requisição foi monitorado no 4Diac, após o número de ocorrências ter atingido 200 o sistema foi parado. Posteriormente foi consultado o número de documentos na coleção em que os dados foram salvos e o resultado como demonstrado na Figura 29 é de 202, assim podendo ser concluído que o sistema é confiável e que nenhum dado foi perdido.

Figura 29: Número de documentos salvos na coleção do MongoDB

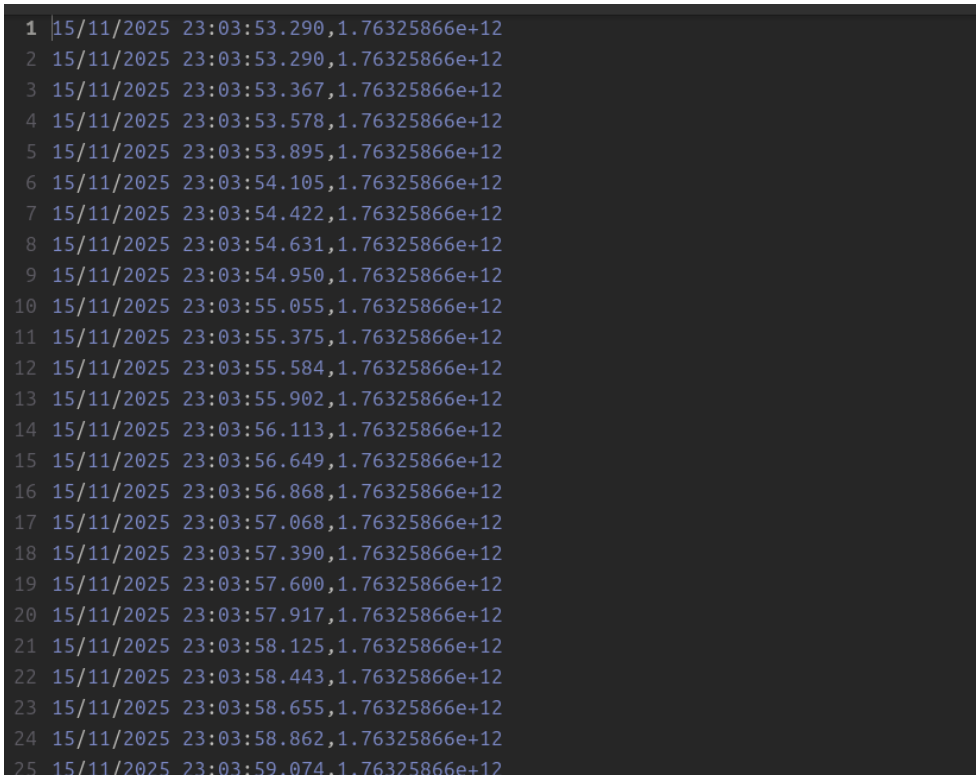



Fonte: Próprio Autor

## 5.5 Teste de Robustez e Desempenho

No teste de robustez, foi testado a consistência do desempenho do bloco MongoDB e comparado com o intervalo de ocorrência do evento de saída IND do bloco de comunicação. A partir do arquivo CSV gerado pelos blocos CRONOMETROFB e CSV_Block, foi observado o desempenho do bloco de banco de dados a partir da diferença de tempo da ocorrência do evento de entrada REQ e o evento de saída CNF. Como mostrado na Figura 30, a coluna de duração permanece com o mesmo valor de 1,76ms enquanto a coluna do tempo mostra o tempo exato em que ocorreu o evento e com isso foi possível calcular as diferenças de tempo entre um evento e outro.

Figura 30: Arquivo CSV

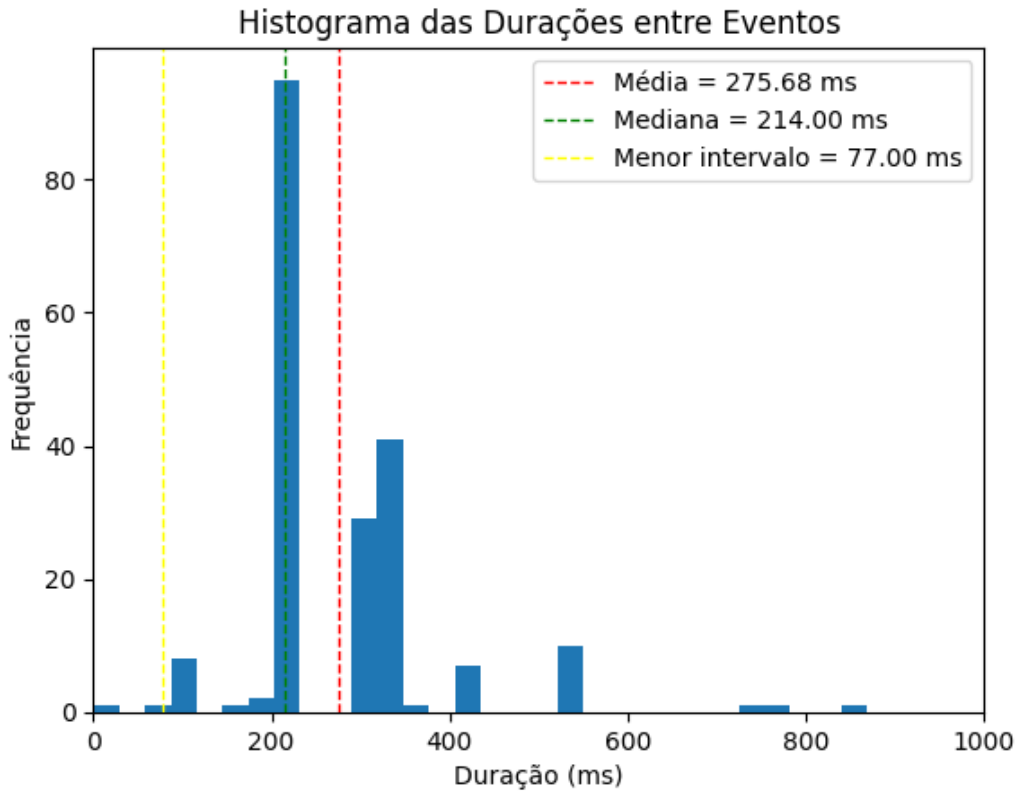


```
15/11/2025 23:03:53.290,1.76325866e+12
15/11/2025 23:03:53.290,1.76325866e+12
15/11/2025 23:03:53.367,1.76325866e+12
15/11/2025 23:03:53.578,1.76325866e+12
15/11/2025 23:03:53.895,1.76325866e+12
15/11/2025 23:03:54.105,1.76325866e+12
15/11/2025 23:03:54.422,1.76325866e+12
15/11/2025 23:03:54.631,1.76325866e+12
15/11/2025 23:03:54.950,1.76325866e+12
15/11/2025 23:03:55.055,1.76325866e+12
15/11/2025 23:03:55.375,1.76325866e+12
15/11/2025 23:03:55.584,1.76325866e+12
15/11/2025 23:03:55.902,1.76325866e+12
15/11/2025 23:03:56.113,1.76325866e+12
15/11/2025 23:03:56.649,1.76325866e+12
15/11/2025 23:03:56.868,1.76325866e+12
15/11/2025 23:03:57.068,1.76325866e+12
15/11/2025 23:03:57.390,1.76325866e+12
15/11/2025 23:03:57.600,1.76325866e+12
15/11/2025 23:03:57.917,1.76325866e+12
15/11/2025 23:03:58.125,1.76325866e+12
15/11/2025 23:03:58.443,1.76325866e+12
15/11/2025 23:03:58.655,1.76325866e+12
15/11/2025 23:03:58.862,1.76325866e+12
15/11/2025 23:03:59.074,1.76325866e+12
```

Fonte: Próprio Autor

Utilizando um script C++, foi possível criar outro arquivo CSV que contém as diferenças de tempo de ocorrência em milissegundos. A partir desse arquivo, realizou-se uma análise de dados usando o gráfico histograma e foi calculado a média e mediana dos dados gerados. Com base no gráfico da Figura 31, observou-se que a média de tempo em que a ESP32 manda os dados é de 275,68ms, a mediana é de 214ms e o menor intervalo de ocorrência é de 77ms. Considerando esses valores, verifica-se que o tempo médio necessário para que o MongoDB armazene um documento é aproximadamente 150 vezes menor que o intervalo de chegada de novos dados e no caso de menor ocorrência chega a ser aproximadamente 40 vezes menor. Portanto, é pouco provável que ocorram perdas de dados por limitações de desempenho do sistema. Esses resultados permitem concluir que o sistema de coleta de dados é robusto e apresenta desempenho adequado para coleta de dados industriais.

Figura 31:Gráfico do intervalo de tempo das coletas de dados




Fonte: Do próprio Autor

## 5.6 Síntese dos Resultados

Os testes realizados permitiram demonstrar que:

- O sistema de coleta de dados baseado na norma IEC 61499 é funcional e eficiente.
- O bloco MongoDB opera de forma estável, permitindo criação de bancos, coleções e inserção de documentos sem falhas.
- A arquitetura distribuída da norma permite integração tanto local quanto remota, validada pelos testes com UaExpert e ESP32.
- O sistema apresenta alto grau de confiabilidade e robustez, atendendo aos requisitos de ambientes industriais.

- O MongoDB se mostrou uma solução adequada para grandes volumes de dados, oferecendo escalabilidade horizontal e flexibilidade estrutural

# 6 CONCLUSÕES E PROPOSTAS DE CONTINUIDADE

Este trabalho apresentou o desenvolvimento de um sistema de coleta de dados baseado na norma IEC 61499, capaz de adquirir informações de equipamentos industriais por meio do protocolo OPC UA e armazená-las em um banco de dados não relacional. O sistema estabelece a conexão e a troca de dados entre um softCLP, executado pelo runtime do 4Diac em ambiente Linux, e um microcontrolador ESP32.

Como demonstrado nos resultados, a adesão à norma IEC 61499 para fazer coleta e armazenamento de dados mostrou-se altamente viável. O sistema de coleta de dados apresentou funcionamento adequado ao cumprir todos os objetivos propostos, além de demonstrar robustez e tolerância a falhas de comunicação. O MongoDB também se mostrou uma alternativa eficiente para o armazenamento em larga escala, atendendo plenamente às demandas de escalabilidade horizontal esperadas em ambientes da Indústria 4.0.

Para projetos futuros, propõe-se a realização de um estudo comparativo de desempenho entre diferentes tipos de bancos de dados, analisando como cada um deles se comporta em projetos baseados na norma IEC 61499. Outra possibilidade de melhoria consiste no desenvolvimento de blocos funcionais capazes de executar as etapas restantes do tratamento de dados, como o pré-processamento e a análise. No âmbito do pré-processamento, podem ser exploradas operações nativas dos bancos de dados, tais como *SELECT (*ou *FIND* em bancos não relacionais), *UPDATE* e *DELETE*.

Durante o desenvolvimento deste trabalho, foi criado um bloco responsável pela comunicação com o banco de dados SQLite 3, cuja função era executar comandos SQL recebidos por meio de um de seus parâmetros. Esse bloco foi capaz de realizar todas as operações previstas no banco de dados, demonstrando que é viável implementar, de forma consistente, outras etapas do tratamento de dados dentro de um ambiente aderente à norma IEC 61499.

No estágio de análise de dados, pode-se empregar o protocolo HTTP para a visualização de gráficos que apresentem o histórico das variáveis monitoradas, por meio de uma integração com scripts desenvolvidos em Python. Outra possibilidade de aprimoramento consiste na criação de blocos capazes de executar algoritmos de

inteligência artificial voltados à análise de dados, ampliando o potencial do sistema para aplicações mais complexas e avançadas.

# REFERÊNCIAS

AWS**.** *O que é NoSQL?* São Paulo: Amazon Web Services, 2025. Disponível em: https://aws.amazon.com/pt/nosql/. Acesso em: 20 dez. 2025.

Arnarson, H., Bremdal, B. A., & Solvang, B. (2022). Reconfigurable Manufacturing: Towards an industrial Big Data approach.

CHRISTENSEN, J. H., STRASSER, T., VALENTINI, A., VYATKIN, V., & ZOITL, A. (2012). The IEC 61499 Function Block Standard: Overview of the Second Edition.

CircuitState Electronics**.** *Getting Started with Espressif ESP32 Wi-Fi & Bluetooth SoC using DOIT ESP32 DevKit V1 Development Board.* CircuitState Electronics, 2023. Disponível em: https://www.circuitstate.com/tutorials/getting-started-with-espressif-esp32-wifi-bluetooth-soc-using-doit-esp32-devkit-v1-development-board/. Acesso em: 20 dez. 2025

DATE, C. J. (1991). *An introduction to database systems. .* Boston: Addison-Wesley Longman Publishing Co.

Diana, M. D., & Gerosa, M. A. (2010). NOSQLnaWeb2.0: UmEstudoComparativo de Bancos. *IX Workshop de Teses e Disserta¸c˜oes em Banco de Dados*.

IEEE - Institute of Electrical and Electronic Engineers. (Dec. de 2012). IEEE Guide for Measuring Earth Resistivity, Ground Impedance, and Earth Surface Potentials of a Grounding System. *IEEE Std 81-2012 (Revision of IEEE Std 81-1983)*, 1-86.

INTERNATIONAL ELECTROTECHNICAL COMMISSION – IEC. *IEC 61131-3: Programmable controllers – Part 3: Programming languages.* 2. ed., 2003. 226 p. Disponível em: https://www.uv.mx/laindustrial/files/2024/01/IEC-61131-3.pdf. Acesso em: 20 dez. 2025

Kajola, P. (2024). IEC 61499 based distributed data collection framework for multivariate time series data.

Kitware. (9 de 10 de 2025). *CMake: The Standard Build System*. Fonte: Cmake: https://cmake.org/features/#system-introspection

Lewis, R., & Zoitl, A. (2014). *Modelling Control Systems Using IEC 61499.*

Liakh, T., Sorokin, R., Akifev, D., Patil, S., & Vyatkin, V. (28 de Julho de 2022). Formal model of IEC 61499 execution trace in FBME IDE.

Merkumians, M. M., Gsellmann, P., & Schitter, G. (2021). Hierarchization and Integration of IEC 61131-3 and IEC 61499 for Enhanced Reusability.

MongoDB Inc. (8 de Outubro de 2025). *Bem vindo ao MongoDB Docs*. Fonte: MongoDB: https://www.mongodb.com/pt-br/docs/

OPC Foundation. (22 de Outubro de 2025). *Unified Architecture – Landingpage*. Fonte: opcfoundation.org: https://opcfoundation.org/about/opc-technologies/opc-ua/

Pang, C., Patil, S., Yang, C.-W., Vyatkin, V., & Shalyto, A. (11 de março de 2015). A Portability Study of IEC 61499: Semantics and Tools.

Pfrommer, J., Ebner, A., Ravikumar, S., & Karunakaran, B. (2018). Open Source OPC UA PubSub over TSN for Realtime Industrial Communication. pp. 1087-1090.

Pinto, L. I. (2014). ICARU-FB: UMA INFRAESTRUTURA DE SOFTWARE ADERENTE À NORMA IEC 61499.

UNIFIED AUTOMATION. (14 de Novembro de 2025). *OPC UA Clients – Downloads*. Fonte: unified-automation: https://www.unified-automation.com/downloads/opc-ua-clients.html

OPEN62541. (14 de Novembro de 2025). *open62541*. Fonte: Github: https://github.com/open62541/open62541

## APÊNDICE A – SCRIPT BD_mongo_project_fbt.cpp

```
/*******************************************************************************
 *** FORTE Library Element
 ***
 *** This file was generated using the 4DIAC FORTE Export Filter V1.0.x NG!
 ***
 *** Name: BD_mongo_project
 *** Description: Service Interface Function Block Type
 *** Version:
 ***     1.0: 2025-11-08/gabriel -  -
 *******************************************************************************/

#include "BD_mongo_project_fbt.h"
#ifdef FORTE_ENABLE_GENERATED_SOURCE_CPP
#include "BD_mongo_project_fbt_gen.cpp"
#endif

#include "core/iec61131_functions.h"
#include "core/datatypes/forte_array_common.h"
#include "core/datatypes/forte_array.h"
#include "core/datatypes/forte_array_fixed.h"
#include "core/datatypes/forte_array_variable.h"
#include "/usr/include/python3.12/Python.h"
#include <iostream>
#include <fstream>
#include <sstream>   // <-- necessário para std::stringstream
#include <string>
#include <vector>
#include <stdio.h>
#include <chrono>
#include <ctime>
#include <iomanip>  // para put_time
```

```
using namespace std;

DEFINE_FIRMWARE_FB(FORTE_BD_mongo_project,
g_nStringIdBD_mongo_project)

const CStringDictionary::TStringId
FORTE_BD_mongo_project::scmDataInputNames[] = {g_nStringIdQI,
g_nStringIdPARAMS, g_nStringIdData1, g_nStringIdData2, g_nStringIdData3};
const CStringDictionary::TStringId
FORTE_BD_mongo_project::scmDataInputTypeIds[] = {g_nStringIdBOOL,
g_nStringIdSTRING, g_nStringIdBOOL, g_nStringIdBOOL, g_nStringIdREAL};
const CStringDictionary::TStringId
FORTE_BD_mongo_project::scmDataOutputNames[] = {g_nStringIdQO,
g_nStringIdSTATUS};
const CStringDictionary::TStringId
FORTE_BD_mongo_project::scmDataOutputTypeIds[] = {g_nStringIdBOOL,
g_nStringIdWSTRING};
const TDataIOID FORTE_BD_mongo_project::scmEIWith[] = {0, 1,
scmWithListDelimiter, 0, 2, 3, 4, 1, scmWithListDelimiter, 0, 2, 3, 4, 1,
scmWithListDelimiter};
const TForteInt16 FORTE_BD_mongo_project::scmEIWithIndexes[] = {0, 3, 9};
const CStringDictionary::TStringId
FORTE_BD_mongo_project::scmEventInputNames[] = {g_nStringIdINIT,
g_nStringIdREQ, g_nStringIdFinish};
const TDataIOID FORTE_BD_mongo_project::scmEOWith[] = {0, 1,
scmWithListDelimiter, 0, 1, scmWithListDelimiter};
const TForteInt16 FORTE_BD_mongo_project::scmEOWithIndexes[] = {0, 3};
const CStringDictionary::TStringId
FORTE_BD_mongo_project::scmEventOutputNames[] = {g_nStringIdINITO,
g_nStringIdCNF};
const SFBInterfaceSpec FORTE_BD_mongo_project::scmFBInterfaceSpec = {
  3, scmEventInputNames, nullptr, scmEIWith, scmEIWithIndexes,
```

```
  2, scmEventOutputNames, nullptr, scmEOWith, scmEOWithIndexes,
  5, scmDataInputNames, scmDataInputTypeIds,
  2, scmDataOutputNames, scmDataOutputTypeIds,
  0, nullptr,
  0, nullptr
};

FORTE_BD_mongo_project::FORTE_BD_mongo_project(const
CStringDictionary::TStringId paInstanceNameId, forte::core::CFBContainer
&paContainer) :
    CFunctionBlock(paContainer, scmFBInterfaceSpec, paInstanceNameId),
    var_QI(0_BOOL),
    var_PARAMS(""_STRING),
    var_Data1(0_BOOL),
    var_Data2(0_BOOL),
    var_Data3(0_REAL),
    var_QO(0_BOOL),
    var_STATUS(u""_WSTRING),
    var_conn_QO(var_QO),
    var_conn_STATUS(var_STATUS),
    conn_INITO(this, 0),
    conn_CNF(this, 1),
    conn_QI(nullptr),
    conn_PARAMS(nullptr),
    conn_Data1(nullptr),
    conn_Data2(nullptr),
    conn_Data3(nullptr),
    conn_QO(this, 0, &var_conn_QO),
    conn_STATUS(this, 1, &var_conn_STATUS) {
};

void FORTE_BD_mongo_project::setInitialValues() {
  var_QI = 0_BOOL;
```

```
  var_PARAMS = ""_STRING;
  var_Data1 = 0_BOOL;
  var_Data2 = 0_BOOL;
  var_Data3 = 0_REAL;
  var_QO = 0_BOOL;
  var_STATUS = u""_WSTRING;
}

string getDateTime() {
    // pega o horário atual
    auto agora = chrono::system_clock::now();

    // converte para time_t
    time_t tempo = chrono::system_clock::to_time_t(agora);

    // converte para struct tm
    tm local_tm = *localtime(&tempo);

    // monta a string formatada
    stringstream ss;
    ss << put_time(&local_tm, "%d/%m/%Y %H:%M:%S");

    return ss.str();
}

void FORTE_BD_mongo_project::executeEvent(const TEventID paEIID,
CEventChainExecutionThread *const paECET) {
  vector<string> campos;
  string campo;
  std::stringstream ss(var_PARAMS.c_str());
  bool Relay0 = static_cast<bool>(var_Data1);
  bool Relay1 = static_cast<bool>(var_Data2);
  float temperature = static_cast<float>(var_Data3);
```

```
switch(paEIID) {
  case scmEventINITID:
       if(var_QI) {
      var_STATUS = u"init_ok"_WSTRING;
    } else {
      var_STATUS = u"init_failed"_WSTRING;
    }
       // Inicializa o interpretador Python
  Py_Initialize();

  if (!Py_IsInitialized()) {
      fprintf(stderr, "Erro ao inicializar Python.\n");
      var_STATUS = u"Erro ao inicializar Python"_WSTRING;
      sendOutputEvent(scmEventINITOID, paECET);
      break;
  }
    sendOutputEvent(scmEventINITOID, paECET);
    break;
  case scmEventREQID:
                while (getline(ss, campo, ';')) {
      campos.push_back(campo);
  }

  char code[512];
  snprintf(code, sizeof(code),
      "from pymongo import MongoClient\n"
      "client = MongoClient('mongodb://localhost:27017/')\n"
      "db = client['%s']\n"
      "colecao = db['%s']\n"
      "colecao.insert_one({'tempo' : '%s', '%s': '%u', '%s': '%u', '%s' : '%.1f'})\n"
      "client.close()",
```

```
campos[0].c_str(),campos[1].c_str(),getDateTime().c_str(),campos[2].c_str(),Relay0,c
ampos[3].c_str(), Relay1, campos[4].c_str(), temperature
    );

    // Executar o código Python
    if (PyRun_SimpleString(code) != 0) {
        std::cerr << "Erro ao executar código Python!" << std::endl;
        var_STATUS = u"Erro ao executar o codigo Python"_WSTRING;
    }else {
    var_STATUS = u"Executando o codigo Python!"_WSTRING;
    }
      sendOutputEvent(scmEventCNFID, paECET);
      break;
    case scmEventFinishID:
      var_STATUS = u"Fechando a API do Python"_WSTRING;
      Py_Finalize();
      sendOutputEvent(scmEventCNFID, paECET);
      break;
  }
}

void FORTE_BD_mongo_project::readInputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITID: {
      readData(0, var_QI, conn_QI);
      readData(1, var_PARAMS, conn_PARAMS);
      break;
    }
    case scmEventREQID: {
      readData(0, var_QI, conn_QI);
      readData(2, var_Data1, conn_Data1);
      readData(3, var_Data2, conn_Data2);
```

```
      readData(4, var_Data3, conn_Data3);
      readData(1, var_PARAMS, conn_PARAMS);
      break;
    }
    case scmEventFinishID: {
      readData(0, var_QI, conn_QI);
      readData(2, var_Data1, conn_Data1);
      readData(3, var_Data2, conn_Data2);
      readData(4, var_Data3, conn_Data3);
      readData(1, var_PARAMS, conn_PARAMS);
      break;
    }
    default:
      break;
  }
}

void FORTE_BD_mongo_project::writeOutputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITOID: {
      writeData(0, var_QO, conn_QO);
      writeData(1, var_STATUS, conn_STATUS);
      break;
    }
    case scmEventCNFID: {
      writeData(0, var_QO, conn_QO);
      writeData(1, var_STATUS, conn_STATUS);
      break;
    }
    default:
      break;
  }
}
```

```
CIEC_ANY *FORTE_BD_mongo_project::getDI(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_QI;
    case 1: return &var_PARAMS;
    case 2: return &var_Data1;
    case 3: return &var_Data2;
    case 4: return &var_Data3;
  }
  return nullptr;
}

CIEC_ANY *FORTE_BD_mongo_project::getDO(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_QO;
    case 1: return &var_STATUS;
  }
  return nullptr;
}

CEventConnection *FORTE_BD_mongo_project::getEOConUnchecked(const
TPortId paIndex) {
  switch(paIndex) {
    case 0: return &conn_INITO;
    case 1: return &conn_CNF;
  }
  return nullptr;
}

CDataConnection **FORTE_BD_mongo_project::getDIConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_QI;
```

```
    case 1: return &conn_PARAMS;
    case 2: return &conn_Data1;
    case 3: return &conn_Data2;
    case 4: return &conn_Data3;
  }
  return nullptr;
}

CDataConnection *FORTE_BD_mongo_project::getDOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_QO;
    case 1: return &conn_STATUS;
  }
  return nullptr;
}
```

## APÊNDICE B – SCRIPT CRONOMETROFB

```
/*******************************************************************************
 *** FORTE Library Element
 ***
 *** This file was generated using the 4DIAC FORTE Export Filter V1.0.x NG!
 ***
 *** Name: cronometroFB
 *** Description: Service Interface Function Block Type
 *** Version:
 ***     1.0: 2025-11-14/gabriel -  -
 *******************************************************************************/

#include "cronometroFB_fbt.h"
#ifdef FORTE_ENABLE_GENERATED_SOURCE_CPP
#include "cronometroFB_fbt_gen.cpp"
#endif

#include "core/iec61131_functions.h"
#include "core/datatypes/forte_array_common.h"
#include "core/datatypes/forte_array.h"
#include "core/datatypes/forte_array_fixed.h"
#include "core/datatypes/forte_array_variable.h"
#include <iostream>
#include <chrono>
#include <fstream>
#include <ctime>

using namespace std;
using namespace std::chrono;

DEFINE_FIRMWARE_FB(FORTE_cronometroFB, g_nStringIdcronometroFB)
```

```
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataInputNames[] =
{g_nStringIdQI};
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataInputTypeIds[] =
{g_nStringIdBOOL};
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataOutputNames[] =
{g_nStringIdSTATUS, g_nStringIdTempo};
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataOutputTypeIds[]
= {g_nStringIdWSTRING, g_nStringIdREAL};
const TDataIOID FORTE_cronometroFB::scmEIWith[] = {0, scmWithListDelimiter, 0,
scmWithListDelimiter, 0, scmWithListDelimiter};
const TForteInt16 FORTE_cronometroFB::scmEIWithIndexes[] = {0, 2, 4};
const CStringDictionary::TStringId FORTE_cronometroFB::scmEventInputNames[] =
{g_nStringIdINIT, g_nStringIdIniciarCronometro, g_nStringIdEncerrarCronometro};
const TDataIOID FORTE_cronometroFB::scmEOWith[] = {1, scmWithListDelimiter, 1,
scmWithListDelimiter, 1, scmWithListDelimiter};
const TForteInt16 FORTE_cronometroFB::scmEOWithIndexes[] = {0, 2, 4};
const CStringDictionary::TStringId FORTE_cronometroFB::scmEventOutputNames[]
= {g_nStringIdINITO, g_nStringIdCronometroIniciado,
g_nStringIdCronometroParado};
const SFBInterfaceSpec FORTE_cronometroFB::scmFBInterfaceSpec = {
  3, scmEventInputNames, nullptr, scmEIWith, scmEIWithIndexes,
  3, scmEventOutputNames, nullptr, scmEOWith, scmEOWithIndexes,
  1, scmDataInputNames, scmDataInputTypeIds,
  2, scmDataOutputNames, scmDataOutputTypeIds,
  0, nullptr,
  0, nullptr
};

FORTE_cronometroFB::FORTE_cronometroFB(const CStringDictionary::TStringId
paInstanceNameId, forte::core::CFBContainer &paContainer) :
    CFunctionBlock(paContainer, scmFBInterfaceSpec, paInstanceNameId),
    var_QI(0_BOOL),
    var_STATUS(u""_WSTRING),
```

```
    var_Tempo(0_REAL),
    var_conn_STATUS(var_STATUS),
    var_conn_Tempo(var_Tempo),
    conn_INITO(this, 0),
    conn_CronometroIniciado(this, 1),
    conn_CronometroParado(this, 2),
    conn_QI(nullptr),
    conn_STATUS(this, 0, &var_conn_STATUS),
    conn_Tempo(this, 1, &var_conn_Tempo) {
};

void FORTE_cronometroFB::setInitialValues() {
  var_QI = 0_BOOL;
  var_STATUS = u""_WSTRING;
  var_Tempo = 0_REAL;
}

void FORTE_cronometroFB::executeEvent(const TEventID paEIID,
CEventChainExecutionThread *const paECET) {

  std::chrono::high_resolution_clock::time_point inicio;
  std::chrono::high_resolution_clock::time_point fim;
  switch(paEIID) {
    case scmEventINITID:
      if(var_QI) {
        var_STATUS = u"init_ok"_WSTRING;
      } else {
        var_STATUS = u"init_failed"_WSTRING;
      }
      sendOutputEvent(scmEventINITOID, paECET);
      break;
    case scmEventIniciarCronometroID:
      inicio = std::chrono::high_resolution_clock::now();
```

```
      sendOutputEvent(scmEventCronometroIniciadoID, paECET);
      break;
    case scmEventEncerrarCronometroID:
      // Marca o tempo final
    fim = std::chrono::high_resolution_clock::now();

    // Calcula duração em milissegundos
    auto duracao = duration_cast<milliseconds>(fim - inicio).count();
    var_Tempo = CIEC_REAL(static_cast<TForteFloat>(duracao));
    sendOutputEvent(scmEventCronometroParadoID, paECET);
      break;
  }
}

void FORTE_cronometroFB::readInputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    case scmEventIniciarCronometroID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    case scmEventEncerrarCronometroID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    default:
      break;
  }
}
```

```
void FORTE_cronometroFB::writeOutputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITOID: {
      writeData(1, var_Tempo, conn_Tempo);
      break;
    }
    case scmEventCronometroIniciadoID: {
      writeData(1, var_Tempo, conn_Tempo);
      break;
    }
    case scmEventCronometroParadoID: {
      writeData(1, var_Tempo, conn_Tempo);
      break;
    }
    default:
      break;
  }
}

CIEC_ANY *FORTE_cronometroFB::getDI(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_QI;
  }
  return nullptr;
}

CIEC_ANY *FORTE_cronometroFB::getDO(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_STATUS;
    case 1: return &var_Tempo;
  }
  return nullptr;
}
```

```
CEventConnection *FORTE_cronometroFB::getEOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_INITO;
    case 1: return &conn_CronometroIniciado;
    case 2: return &conn_CronometroParado;
  }
  return nullptr;
}

CDataConnection **FORTE_cronometroFB::getDIConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_QI;
  }
  return nullptr;
}

CDataConnection *FORTE_cronometroFB::getDOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_STATUS;
    case 1: return &conn_Tempo;
  }
  return nullptr;
}
```

## APÊNDICE C – SCRIPT MYBD_fbt.cpp

```
/*******************************************************************************
 *** FORTE Library Element
 ***
 *** This file was generated using the 4DIAC FORTE Export Filter V1.0.x NG!
 ***
 *** Name: MyBD
 *** Description: Service Interface Function Block Type
 *** Version:
 ***     1.0: 2025-08-19/gabriel -  -
 *******************************************************************************/

#include "MyBD_fbt.h"
#ifdef FORTE_ENABLE_GENERATED_SOURCE_CPP
#include "MyBD_fbt_gen.cpp"
#endif

#include "core/iec61131_functions.h"
#include "core/datatypes/forte_array_common.h"
#include "core/datatypes/forte_array.h"
#include "core/datatypes/forte_array_fixed.h"
#include "core/datatypes/forte_array_variable.h"
#include "sqlite3.h"
#include <iostream>

DEFINE_FIRMWARE_FB(FORTE_MyBD, g_nStringIdMyBD)

const CStringDictionary::TStringId FORTE_MyBD::scmDataInputNames[] =
{g_nStringIdQI, g_nStringIdPARAMS, g_nStringIdQUERY};
const CStringDictionary::TStringId FORTE_MyBD::scmDataInputTypeIds[] =
{g_nStringIdBOOL, g_nStringIdWSTRING, g_nStringIdWSTRING};
```

```
const CStringDictionary::TStringId FORTE_MyBD::scmDataOutputNames[] =
{g_nStringIdSTATUS, g_nStringIdRD};
const CStringDictionary::TStringId FORTE_MyBD::scmDataOutputTypeIds[] =
{g_nStringIdWSTRING, g_nStringIdWSTRING};
const TDataIOID FORTE_MyBD::scmEIWith[] = {0, 1, scmWithListDelimiter, 0, 1,
scmWithListDelimiter, 0, 2, 1, scmWithListDelimiter, 0, 1, 2, scmWithListDelimiter};
const TForteInt16 FORTE_MyBD::scmEIWithIndexes[] = {0, 3, 6, 10};
const CStringDictionary::TStringId FORTE_MyBD::scmEventInputNames[] =
{g_nStringIdINIT, g_nStringIdCONNECTION, g_nStringIdQUERY_ACTION,
g_nStringIdREAD_DATA};
const TDataIOID FORTE_MyBD::scmEOWith[] = {0, scmWithListDelimiter, 0,
scmWithListDelimiter, 0, 1, scmWithListDelimiter};
const TForteInt16 FORTE_MyBD::scmEOWithIndexes[] = {0, 2, 4};
const CStringDictionary::TStringId FORTE_MyBD::scmEventOutputNames[] =
{g_nStringIdINITO, g_nStringIdCNF, g_nStringIdFILE_OUT};
const SFBInterfaceSpec FORTE_MyBD::scmFBInterfaceSpec = {
  4, scmEventInputNames, nullptr, scmEIWith, scmEIWithIndexes,
  3, scmEventOutputNames, nullptr, scmEOWith, scmEOWithIndexes,
  3, scmDataInputNames, scmDataInputTypeIds,
  2, scmDataOutputNames, scmDataOutputTypeIds,
  0, nullptr,
  0, nullptr
};

FORTE_MyBD::FORTE_MyBD(const CStringDictionary::TStringId
paInstanceNameId, forte::core::CFBContainer &paContainer) :
    CFunctionBlock(paContainer, scmFBInterfaceSpec, paInstanceNameId),
    var_QI(0_BOOL),
    var_PARAMS(u""_WSTRING),
    var_QUERY(u""_WSTRING),
    var_STATUS(u""_WSTRING),
    var_RD(u""_WSTRING),
    var_conn_STATUS(var_STATUS),
```

```
    var_conn_RD(var_RD),
    conn_INITO(this, 0),
    conn_CNF(this, 1),
    conn_FILE_OUT(this, 2),
    conn_QI(nullptr),
    conn_PARAMS(nullptr),
    conn_QUERY(nullptr),
    conn_STATUS(this, 0, &var_conn_STATUS),
    conn_RD(this, 1, &var_conn_RD) {
};

int callback(void* NotUsed, int argc, char** argv, char** azColName) {
    std::string* result = reinterpret_cast<std::string*>(NotUsed);
    for (int i = 0; i < argc; i++) {
        std::cout << azColName[i] << ": " << (argv[i] ? argv[i] : "NULL") << "\t";
    }
    std::cout << "\n";
     if (argc > 0 && argv[0]) {
        *result = argv[0];  // pega o primeiro valor da primeira coluna
    }
    return 0;
}

void FORTE_MyBD::setInitialValues() {
  var_QI = 0_BOOL;
  var_PARAMS = u""_WSTRING;
  var_QUERY = u""_WSTRING;
  var_STATUS = u""_WSTRING;
  var_RD = u""_WSTRING;
}

void FORTE_MyBD::executeEvent(const TEventID paEIID,
CEventChainExecutionThread *const paECET) {
```

```
    sqlite3 *db;
    //sqlite3_stmt* stmt;
    int rc;
    char *errMsg = nullptr;
    std::string resultado;
    // Suponha que a saída 1 seja do tipo WSTRING
    CIEC_WSTRING *outWStr = static_cast<CIEC_WSTRING*>(getDO(1));

  switch(paEIID) {
    case scmEventINITID:
      if(var_QI) {
        var_STATUS = u"init_ok"_WSTRING;
      } else {
        var_STATUS = u"init_failed"_WSTRING;
      }
      sendOutputEvent(scmEventINITOID, paECET);
      break;
    case scmEventCONNECTIONID:


    rc = sqlite3_open(var_PARAMS.getValue(), &db);

        if (rc != SQLITE_OK) {
        fprintf(stderr, "Erro ao abrir o banco de dados: %s\n", sqlite3_errmsg(db));
        var_STATUS = u"connection_failed"_WSTRING;
        sendOutputEvent(scmEventCNFID, paECET);
        break;
    }

    std::cout << "Banco de dados aberto com sucesso: " << var_PARAMS.getValue()
<< std::endl;
    var_STATUS = u"connection_sucess"_WSTRING;
    sqlite3_close(db);
```

```
    sendOutputEvent(scmEventCNFID, paECET);
    break;
  case scmEventQUERY_ACTIONID:

    rc = sqlite3_open(var_PARAMS.getValue(), &db);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Erro ao abrir o banco de dados: %s\n", sqlite3_errmsg(db));
        var_STATUS = u"connection_failed"_WSTRING;
        sendOutputEvent(scmEventCNFID, paECET);
        break;
  }

     // Executar query
  if (sqlite3_exec(db, var_QUERY.getValue(), callback, nullptr, &errMsg) !=
SQLITE_OK) {
       std::cerr << "Erro ao executar query: " << errMsg << std::endl;
      var_STATUS = u"QUERY_failed"_WSTRING;
      sqlite3_free(errMsg);
      sqlite3_close(db);
      sendOutputEvent(scmEventCNFID, paECET);
      break;
  }
    var_STATUS = u"QUERY_ok"_WSTRING;
    sqlite3_close(db);
    sendOutputEvent(scmEventCNFID, paECET);
    break;

    case scmEventREAD_DATAID:

    rc = sqlite3_open(var_PARAMS.getValue(), &db);
```

```
      if (sqlite3_exec(db, var_QUERY.getValue(), callback, &resultado, &errMsg) !=
SQLITE_OK) {
        std::cerr << "Erro ao executar query: " << errMsg << std::endl;
        var_STATUS = u"QUERY_failed"_WSTRING;
        sqlite3_free(errMsg);
        sqlite3_close(db);
        sendOutputEvent(scmEventFILE_OUTID, paECET);
        break;
    } else {
      std::cout << "nomes:" << resultado << std::endl;
      }


      var_STATUS = u"QUERY_ok"_WSTRING;
      // atribui diretamente a partir de std::string
      //var_RD.clear();
      //var_RD.fromString(resultado.c_str());
      //var_RD = u"teste"_WSTRING;

      // Alterando o valor
      outWStr->fromString(resultado.c_str());



      sendOutputEvent(scmEventFILE_OUTID, paECET);
      break;
  }
}

void FORTE_MyBD::readInputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITID: {
      readData(0, var_QI, conn_QI);
```

```
      readData(1, var_PARAMS, conn_PARAMS);
      break;
    }
    case scmEventCONNECTIONID: {
      readData(0, var_QI, conn_QI);
      readData(1, var_PARAMS, conn_PARAMS);
      break;
    }
    case scmEventQUERY_ACTIONID: {
      readData(0, var_QI, conn_QI);
      readData(2, var_QUERY, conn_QUERY);
      readData(1, var_PARAMS, conn_PARAMS);
      break;
    }
    case scmEventREAD_DATAID: {
      readData(0, var_QI, conn_QI);
      readData(1, var_PARAMS, conn_PARAMS);
      readData(2, var_QUERY, conn_QUERY);
      break;
    }
    default:
      break;
  }
}

void FORTE_MyBD::writeOutputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITOID: {
      writeData(0, var_STATUS, conn_STATUS);
      break;
    }
    case scmEventCNFID: {
      writeData(0, var_STATUS, conn_STATUS);
```

```
      break;
    }
    case scmEventFILE_OUTID: {
      writeData(0, var_STATUS, conn_STATUS);
      writeData(1, var_RD, conn_RD);
      break;
    }
    default:
      break;
  }
}

CIEC_ANY *FORTE_MyBD::getDI(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_QI;
    case 1: return &var_PARAMS;
    case 2: return &var_QUERY;
  }
  return nullptr;
}

CIEC_ANY *FORTE_MyBD::getDO(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_STATUS;
    case 1: return &var_RD;
  }
  return nullptr;
}

CEventConnection *FORTE_MyBD::getEOConUnchecked(const TPortId paIndex) {
  switch(paIndex) {
    case 0: return &conn_INITO;
    case 1: return &conn_CNF;
```

```
    case 2: return &conn_FILE_OUT;
  }
  return nullptr;
}

CDataConnection **FORTE_MyBD::getDIConUnchecked(const TPortId paIndex) {
  switch(paIndex) {
    case 0: return &conn_QI;
    case 1: return &conn_PARAMS;
    case 2: return &conn_QUERY;
  }
  return nullptr;
}

CDataConnection *FORTE_MyBD::getDOConUnchecked(const TPortId paIndex) {
  switch(paIndex) {
    case 0: return &conn_STATUS;
    case 1: return &conn_RD;
  }
  return nullptr;
}
```

## APÊNDICE D – SCRIPT CSV_BLOCK

```
/*******************************************************************************
 *** FORTE Library Element
 ***
 *** This file was generated using the 4DIAC FORTE Export Filter V1.0.x NG!
 ***
 *** Name: cronometroFB
 *** Description: Service Interface Function Block Type
 *** Version:
 ***     1.0: 2025-11-14/gabriel -  -
 *******************************************************************************/

#include "cronometroFB_fbt.h"
#ifdef FORTE_ENABLE_GENERATED_SOURCE_CPP
#include "cronometroFB_fbt_gen.cpp"
#endif

#include "core/iec61131_functions.h"
#include "core/datatypes/forte_array_common.h"
#include "core/datatypes/forte_array.h"
#include "core/datatypes/forte_array_fixed.h"
#include "core/datatypes/forte_array_variable.h"
#include <iostream>
#include <chrono>
#include <fstream>
#include <ctime>

using namespace std;
using namespace std::chrono;

DEFINE_FIRMWARE_FB(FORTE_cronometroFB, g_nStringIdcronometroFB)
```

```
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataInputNames[] =
{g_nStringIdQI};
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataInputTypeIds[] =
{g_nStringIdBOOL};
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataOutputNames[] =
{g_nStringIdSTATUS, g_nStringIdTempo};
const CStringDictionary::TStringId FORTE_cronometroFB::scmDataOutputTypeIds[]
= {g_nStringIdWSTRING, g_nStringIdREAL};
const TDataIOID FORTE_cronometroFB::scmEIWith[] = {0, scmWithListDelimiter, 0,
scmWithListDelimiter, 0, scmWithListDelimiter};
const TForteInt16 FORTE_cronometroFB::scmEIWithIndexes[] = {0, 2, 4};
const CStringDictionary::TStringId FORTE_cronometroFB::scmEventInputNames[] =
{g_nStringIdINIT, g_nStringIdIniciarCronometro, g_nStringIdEncerrarCronometro};
const TDataIOID FORTE_cronometroFB::scmEOWith[] = {1, scmWithListDelimiter, 1,
scmWithListDelimiter, 1, scmWithListDelimiter};
const TForteInt16 FORTE_cronometroFB::scmEOWithIndexes[] = {0, 2, 4};
const CStringDictionary::TStringId FORTE_cronometroFB::scmEventOutputNames[]
= {g_nStringIdINITO, g_nStringIdCronometroIniciado,
g_nStringIdCronometroParado};
const SFBInterfaceSpec FORTE_cronometroFB::scmFBInterfaceSpec = {
  3, scmEventInputNames, nullptr, scmEIWith, scmEIWithIndexes,
  3, scmEventOutputNames, nullptr, scmEOWith, scmEOWithIndexes,
  1, scmDataInputNames, scmDataInputTypeIds,
  2, scmDataOutputNames, scmDataOutputTypeIds,
  0, nullptr,
  0, nullptr
};

FORTE_cronometroFB::FORTE_cronometroFB(const CStringDictionary::TStringId
paInstanceNameId, forte::core::CFBContainer &paContainer) :
    CFunctionBlock(paContainer, scmFBInterfaceSpec, paInstanceNameId),
```

```
    var_QI(0_BOOL),
    var_STATUS(u""_WSTRING),
    var_Tempo(0_REAL),
    var_conn_STATUS(var_STATUS),
    var_conn_Tempo(var_Tempo),
    conn_INITO(this, 0),
    conn_CronometroIniciado(this, 1),
    conn_CronometroParado(this, 2),
    conn_QI(nullptr),
    conn_STATUS(this, 0, &var_conn_STATUS),
    conn_Tempo(this, 1, &var_conn_Tempo) {
};

void FORTE_cronometroFB::setInitialValues() {
  var_QI = 0_BOOL;
  var_STATUS = u""_WSTRING;
  var_Tempo = 0_REAL;
}

void FORTE_cronometroFB::executeEvent(const TEventID paEIID,
CEventChainExecutionThread *const paECET) {

  std::chrono::high_resolution_clock::time_point inicio;
  std::chrono::high_resolution_clock::time_point fim;
  switch(paEIID) {
    case scmEventINITID:
      if(var_QI) {
        var_STATUS = u"init_ok"_WSTRING;
      } else {
        var_STATUS = u"init_failed"_WSTRING;
      }
      sendOutputEvent(scmEventINITOID, paECET);
```

```
      break;
    case scmEventIniciarCronometroID:
      inicio = std::chrono::high_resolution_clock::now();
      sendOutputEvent(scmEventCronometroIniciadoID, paECET);
      break;
    case scmEventEncerrarCronometroID:
      // Marca o tempo final
    fim = std::chrono::high_resolution_clock::now();

    // Calcula duração em milissegundos
    auto duracao = duration_cast<milliseconds>(fim - inicio).count();
    var_Tempo = CIEC_REAL(static_cast<TForteFloat>(duracao));
    sendOutputEvent(scmEventCronometroParadoID, paECET);
      break;
  }
}

void FORTE_cronometroFB::readInputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    case scmEventIniciarCronometroID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    case scmEventEncerrarCronometroID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    default:
```

```
      break;
  }
}

void FORTE_cronometroFB::writeOutputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITOID: {
      writeData(1, var_Tempo, conn_Tempo);
      break;
    }
    case scmEventCronometroIniciadoID: {
      writeData(1, var_Tempo, conn_Tempo);
      break;
    }
    case scmEventCronometroParadoID: {
      writeData(1, var_Tempo, conn_Tempo);
      break;
    }
    default:
      break;
  }
}

CIEC_ANY *FORTE_cronometroFB::getDI(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_QI;
  }
  return nullptr;
}

CIEC_ANY *FORTE_cronometroFB::getDO(const size_t paIndex) {
  switch(paIndex) {
```

```
    case 0: return &var_STATUS;
    case 1: return &var_Tempo;
  }
  return nullptr;
}

CEventConnection *FORTE_cronometroFB::getEOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_INITO;
    case 1: return &conn_CronometroIniciado;
    case 2: return &conn_CronometroParado;
  }
  return nullptr;
}

CDataConnection **FORTE_cronometroFB::getDIConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_QI;
  }
  return nullptr;
}

CDataConnection *FORTE_cronometroFB::getDOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_STATUS;
    case 1: return &conn_Tempo;
  }
  return nullptr;
}
```

## APÊNDICE E – SCRIPT pythonBlock_fbt.cpp

```
/*******************************************************************************
 *** FORTE Library Element
 ***
 *** This file was generated using the 4DIAC FORTE Export Filter V1.0.x NG!
 ***
 *** Name: pythonBlock
 *** Description: Service Interface Function Block Type
 *** Version:
 ***     1.0: 2025-06-05/gabriel -  -
 *******************************************************************************/

#include "pythonBlock_fbt.h"
#ifdef FORTE_ENABLE_GENERATED_SOURCE_CPP
#include "pythonBlock_fbt_gen.cpp"
#endif

#include "core/iec61131_functions.h"
#include "core/datatypes/forte_array_common.h"
#include "core/datatypes/forte_array.h"
#include "core/datatypes/forte_array_fixed.h"
#include "core/datatypes/forte_array_variable.h"

DEFINE_FIRMWARE_FB(FORTE_pythonBlock, g_nStringIdpythonBlock)

const CStringDictionary::TStringId FORTE_pythonBlock::scmDataInputNames[] =
{g_nStringIdQI, g_nStringIdFILE_PATH};
const CStringDictionary::TStringId FORTE_pythonBlock::scmDataInputTypeIds[] =
{g_nStringIdBOOL, g_nStringIdWSTRING};
const CStringDictionary::TStringId FORTE_pythonBlock::scmDataOutputNames[] =
{g_nStringIdSTATUS, g_nStringIdDO1};
```

```
const CStringDictionary::TStringId FORTE_pythonBlock::scmDataOutputTypeIds[] =
{g_nStringIdWSTRING, g_nStringIdBOOL};
const TDataIOID FORTE_pythonBlock::scmEIWith[] = {0, scmWithListDelimiter, 0, 1,
scmWithListDelimiter};
const TForteInt16 FORTE_pythonBlock::scmEIWithIndexes[] = {0, 2};
const CStringDictionary::TStringId FORTE_pythonBlock::scmEventInputNames[] =
{g_nStringIdINIT, g_nStringIdREQ};
const TDataIOID FORTE_pythonBlock::scmEOWith[] = {0, scmWithListDelimiter, 0,
scmWithListDelimiter};
const TForteInt16 FORTE_pythonBlock::scmEOWithIndexes[] = {0, 2};
const CStringDictionary::TStringId FORTE_pythonBlock::scmEventOutputNames[] =
{g_nStringIdINITO, g_nStringIdCNF};
const SFBInterfaceSpec FORTE_pythonBlock::scmFBInterfaceSpec = {
  2, scmEventInputNames, nullptr, scmEIWith, scmEIWithIndexes,
  2, scmEventOutputNames, nullptr, scmEOWith, scmEOWithIndexes,
  2, scmDataInputNames, scmDataInputTypeIds,
  2, scmDataOutputNames, scmDataOutputTypeIds,
  0, nullptr,
  0, nullptr
};

FORTE_pythonBlock::FORTE_pythonBlock(const CStringDictionary::TStringId
paInstanceNameId, forte::core::CFBContainer &paContainer) :
    CFunctionBlock(paContainer, scmFBInterfaceSpec, paInstanceNameId),
    var_QI(0_BOOL),
    var_FILE_PATH(u""_WSTRING),
    var_STATUS(u""_WSTRING),
    var_DO1(0_BOOL),
    var_conn_STATUS(var_STATUS),
    var_conn_DO1(var_DO1),
    conn_INITO(this, 0),
    conn_CNF(this, 1),
```

```
    conn_QI(nullptr),
    conn_FILE_PATH(nullptr),
    conn_STATUS(this, 0, &var_conn_STATUS),
    conn_DO1(this, 1, &var_conn_DO1) {
};

void FORTE_pythonBlock::setInitialValues() {
  var_QI = 0_BOOL;
  var_FILE_PATH = u""_WSTRING;
  var_STATUS = u""_WSTRING;
  var_DO1 = 0_BOOL;
}

void FORTE_pythonBlock::executeEvent(const TEventID paEIID,
CEventChainExecutionThread *const paECET) {
  switch(paEIID) {
  case scmEventINITID:
      if(var_QI) {
        var_STATUS = u"init_ok"_WSTRING;
      } else {
        var_STATUS = u"init_failed"_WSTRING;
      }
      sendOutputEvent(scmEventINITOID, paECET);
      break;
    case scmEventREQID:
      if(var_QI){
      system(var_FILE_PATH.getValue());
      }
      sendOutputEvent(scmEventCNFID, paECET);
      break;
  }
}
```

```
void FORTE_pythonBlock::readInputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITID: {
      readData(0, var_QI, conn_QI);
      break;
    }
    case scmEventREQID: {
      readData(0, var_QI, conn_QI);
      readData(1, var_FILE_PATH, conn_FILE_PATH);
      break;
    }
    default:
      break;
  }
}

void FORTE_pythonBlock::writeOutputData(const TEventID paEIID) {
  switch(paEIID) {
    case scmEventINITOID: {
      writeData(0, var_STATUS, conn_STATUS);
      break;
    }
    case scmEventCNFID: {
      writeData(0, var_STATUS, conn_STATUS);
      break;
    }
    default:
      break;
  }
}
```

```
CIEC_ANY *FORTE_pythonBlock::getDI(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_QI;
    case 1: return &var_FILE_PATH;
  }
  return nullptr;
}

CIEC_ANY *FORTE_pythonBlock::getDO(const size_t paIndex) {
  switch(paIndex) {
    case 0: return &var_STATUS;
    case 1: return &var_DO1;
  }
  return nullptr;
}

CEventConnection *FORTE_pythonBlock::getEOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_INITO;
    case 1: return &conn_CNF;
  }
  return nullptr;
}

CDataConnection **FORTE_pythonBlock::getDIConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_QI;
    case 1: return &conn_FILE_PATH;
  }
  return nullptr;
```

```
}

CDataConnection *FORTE_pythonBlock::getDOConUnchecked(const TPortId
paIndex) {
  switch(paIndex) {
    case 0: return &conn_STATUS;
    case 1: return &conn_DO1;
  }
  return nullptr;
}
```