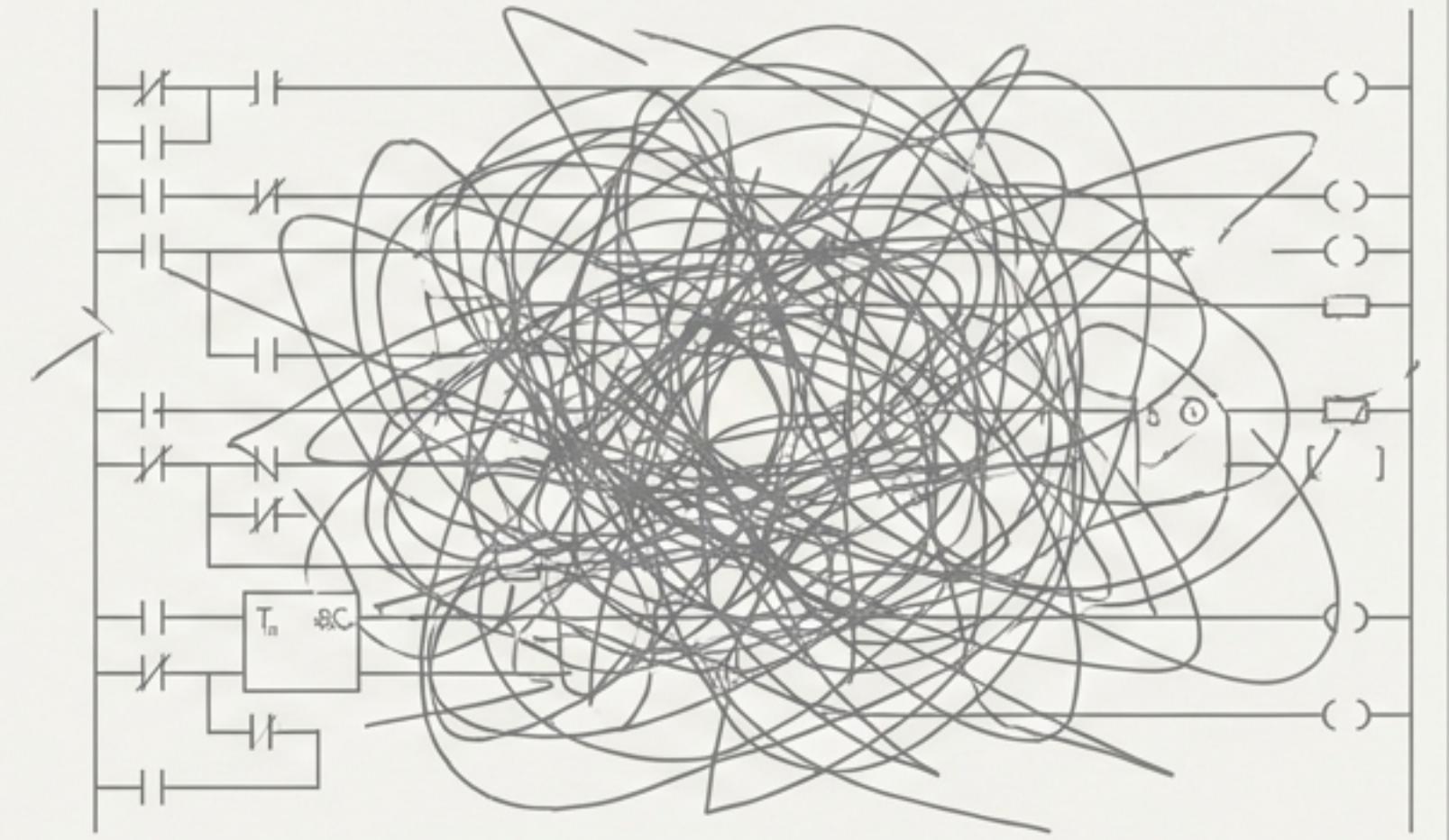
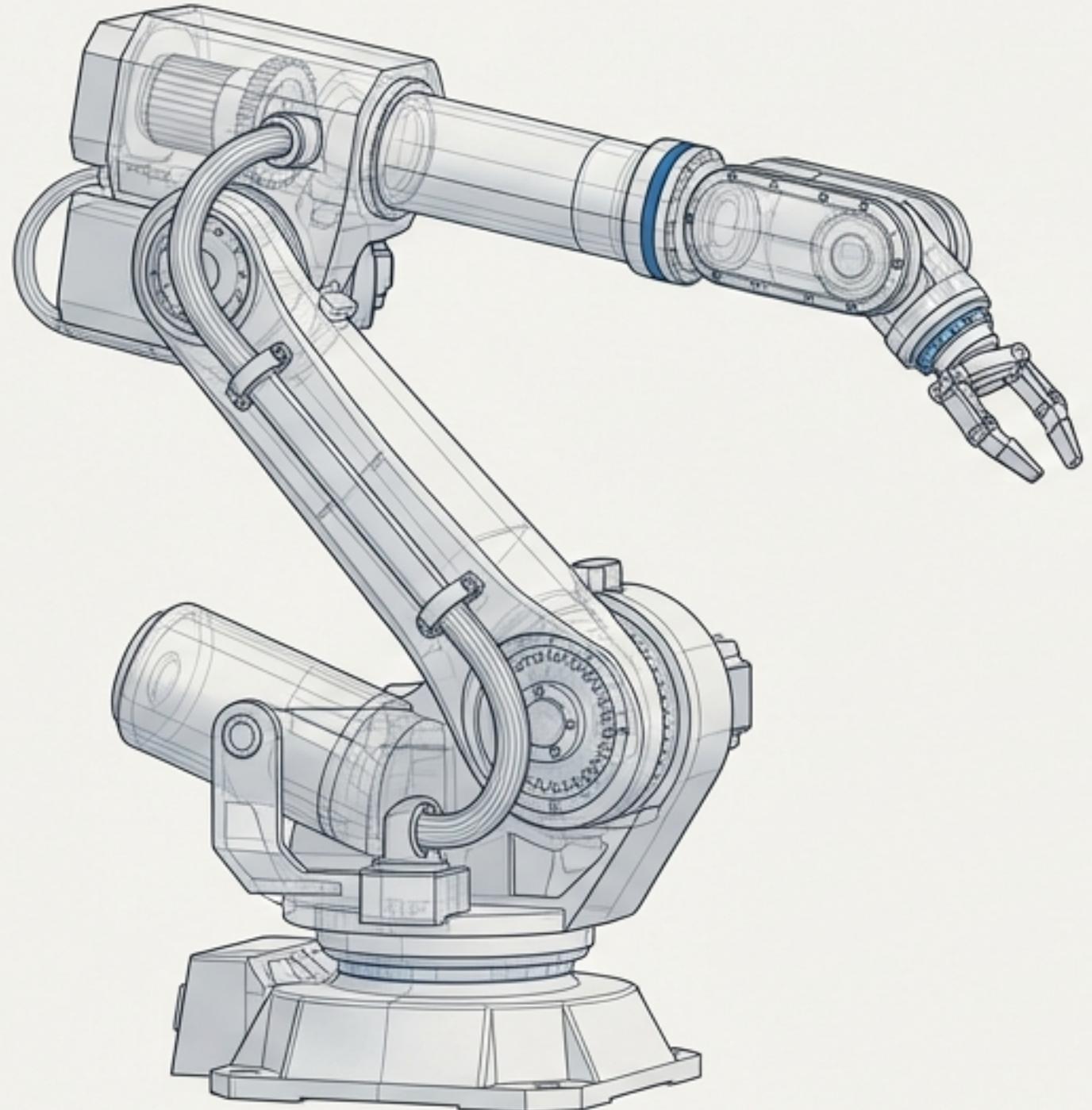
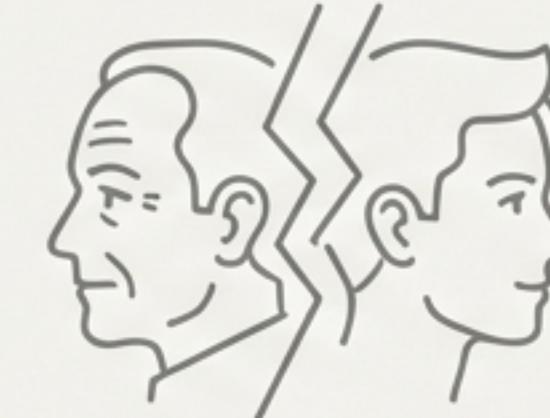
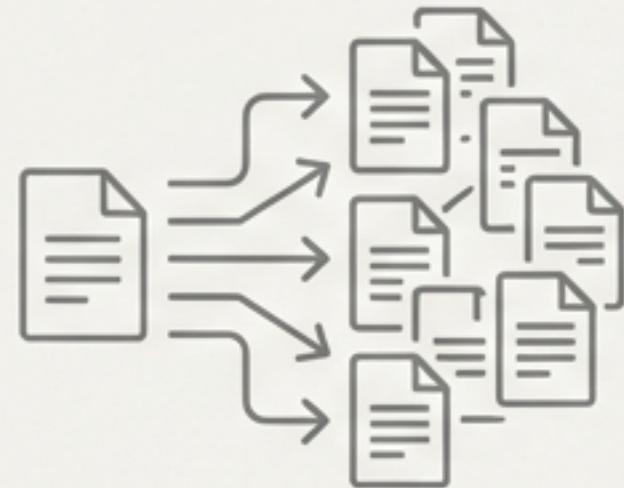


# ハードウェアは進化したが、 制御ソフトウェア開発は取り残されていないか？



製造現場の要求は、年々高度化・複雑化している。  
演算能力、通信速度、ハードウェアの選択肢は飛躍的に向上した。  
しかし、その中核を担う制御ソフトウェアの開発・保守手法は、30年前のパラダイムから本当に進化できているだろうか？  
本質的な課題は、ハードウェアのコストではなく、ソフトウェアのライフサイクルコストにある。

# ソフトウェア開発の「苦しみ」が、生産性の足枷となっている



## ① 装置バリエーションの管理コスト増大

似て非なる装置ごとに、ラダープログラムが乱立。仕様変更のたびに、全てのプログラムへ修正を「コピペ」して回る非効率な作業が発生。バグの温床となり、管理工数は膨れ上がる一方。

「コピペ開発の地獄」

## ② 複雑化する要求とラダー言語の限界

画像処理やAI連携、高度なデータ処理など、現代の要求はラダー言語の表現力を超えている。結果として、プログラムはスパゲッティコード化し、解読も修正も困難に。

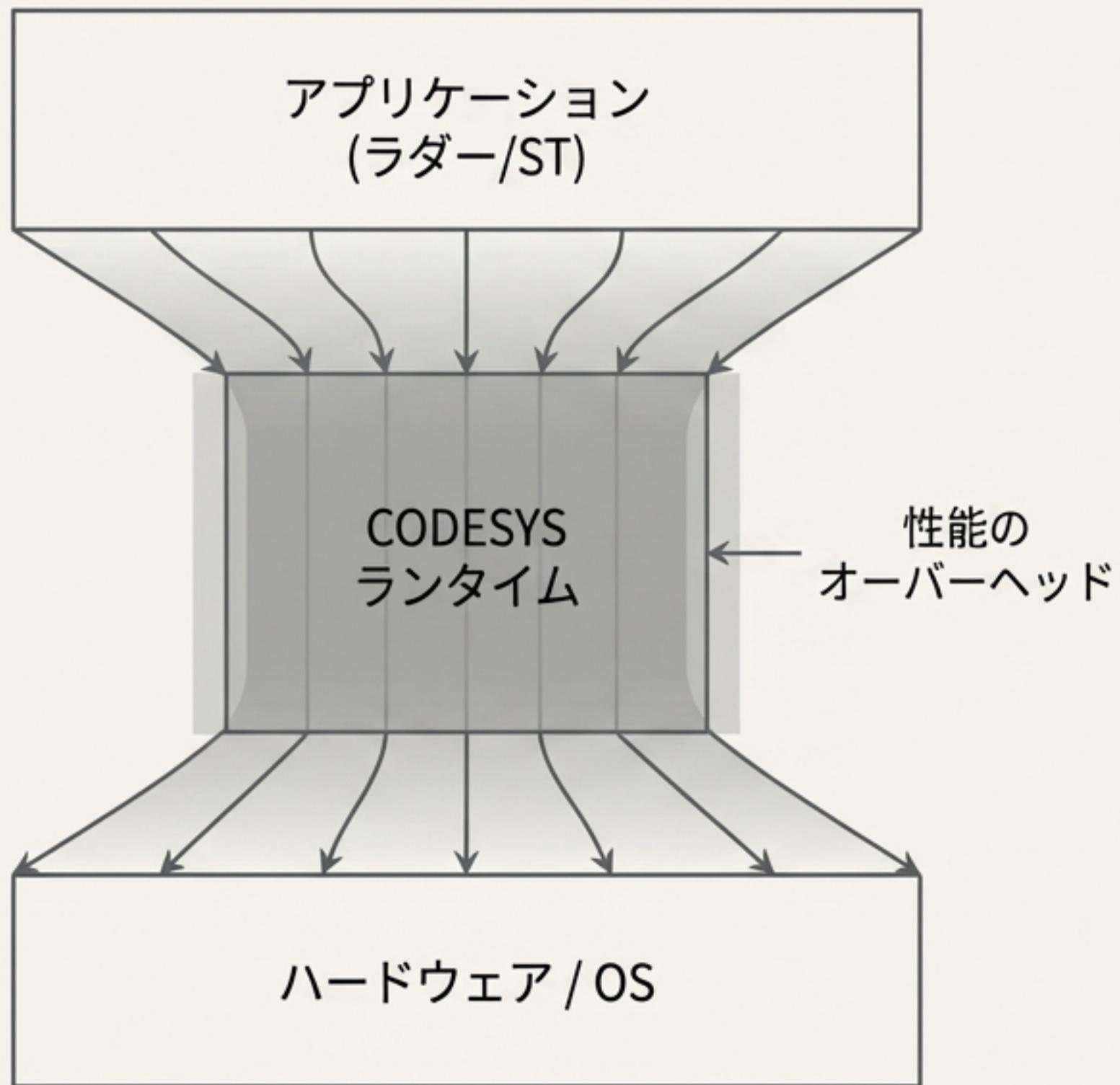
「スパゲッティコード化」

## ③ 専門技術者の不足と技術継承の断絶

「ラダーが書けるベテラン」は高齢化し、その確保は年々困難になっている。一方で、C++ やC#を扱える若手ITエンジニアはいても、独自の開発環境が参入障壁となり、その能力を活かせない。

「ベテラン依存からの脱却」

# 既存のソフトウェアPLCは、根本的な解決策ではない



市場の標準であるCODESYSのようなソフトウェアPLCは、「PCをPLCとして振る舞わせる」アプローチを採用。これは、ハードウェアの違いを吸収する「ランタイム」を介して動作する。

## 本質的な制約:

- **開発パラダイムの旧態依然:** プログラムの書き方はIEC 61131-3規格に縛られ、「コピペ開発」や「バージョン管理の難しさ」といったソフトウェア開発の苦しみは解決されない。
- **性能のオーバーヘッド:** 常にランタイムを介するため、PC本来の演算性能を限界まで引き出すことができず、オーバーヘッドが発生する。
- **IT技術との分断:** 最新のIT技術を適用するには、外部関数呼び出しなどの追加実装が必要となり、開発フローが分断される。

# 新しいカテゴリ：Software Defined Automation Framework

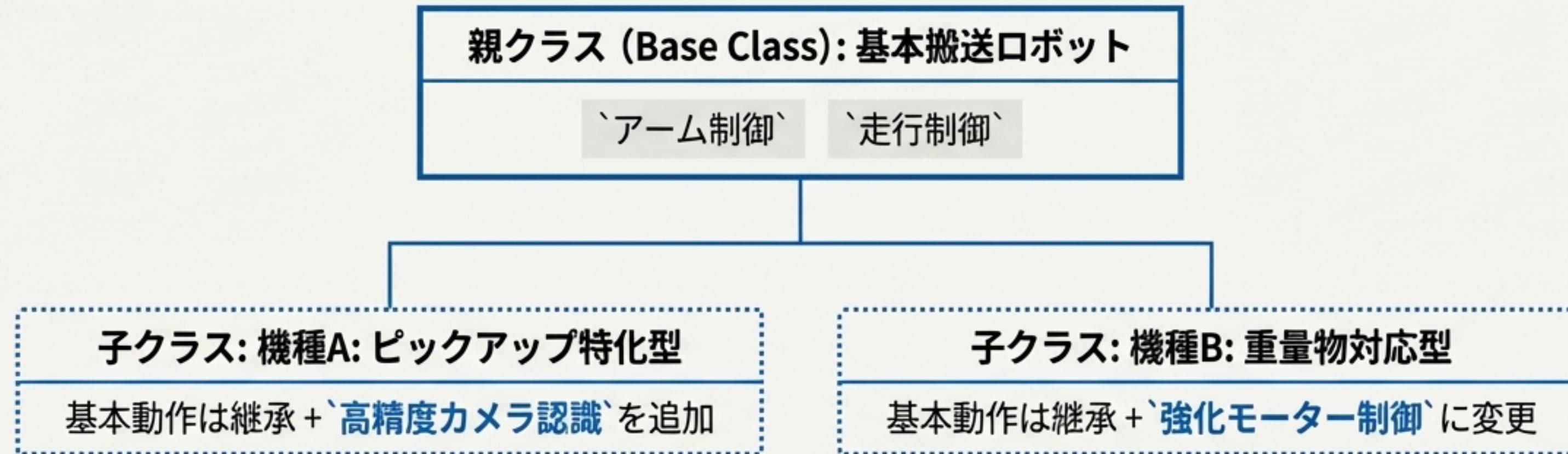
Object-Component Automation (OCA)



OCAは、単なるPC上のPLCエミュレーターではない。  
「ITの表現力」と「OTの堅牢性」をアーキテクチャレベルで融合させるために、  
ゼロから設計された次世代の制御プラットフォーム。  
その目的は、PCのポテンシャルを制御に直結させ、PLCを超える性能と、現代  
的なソフトウェア開発手法による圧倒的な生産性を両立することにある。

# 「継承」の力で、ソフトウェア資産の価値を最大化する

制御ロジックを、再利用可能な「オブジェクト」として構築する。



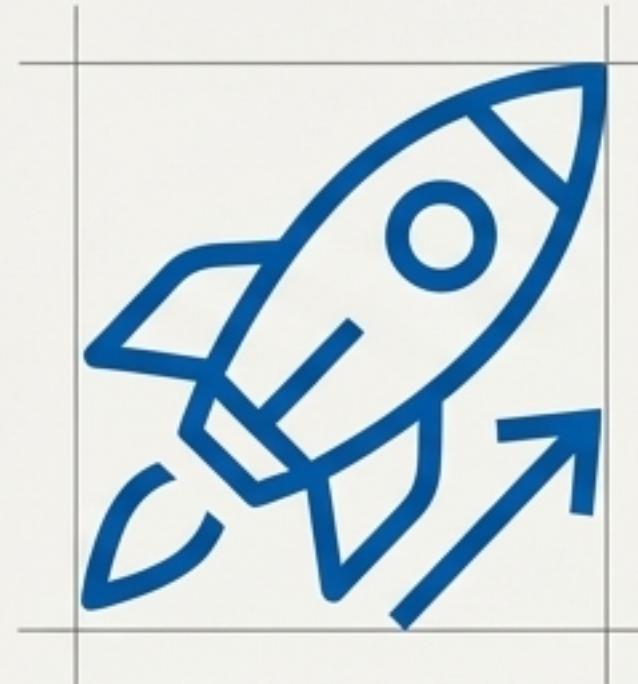
Before (従来の問題)	After (OCAの解決策)
機種A、B、C...それぞれに独立したラダープログラムが存在。 基本動作の修正は、すべてのファイルにコピペが必要。	「基本動作（親クラス）」を一度修正すれば、派生する全ての機種（子クラス）に自動で反映される。 これにより、コピペ開発から脱却し、派生機種のソフトウェア管理工数を劇的に削減する。

A

B

C

# OCAが提供する3つの核となる価値



## Performance & Portability

### 圧倒的な性能と移植性

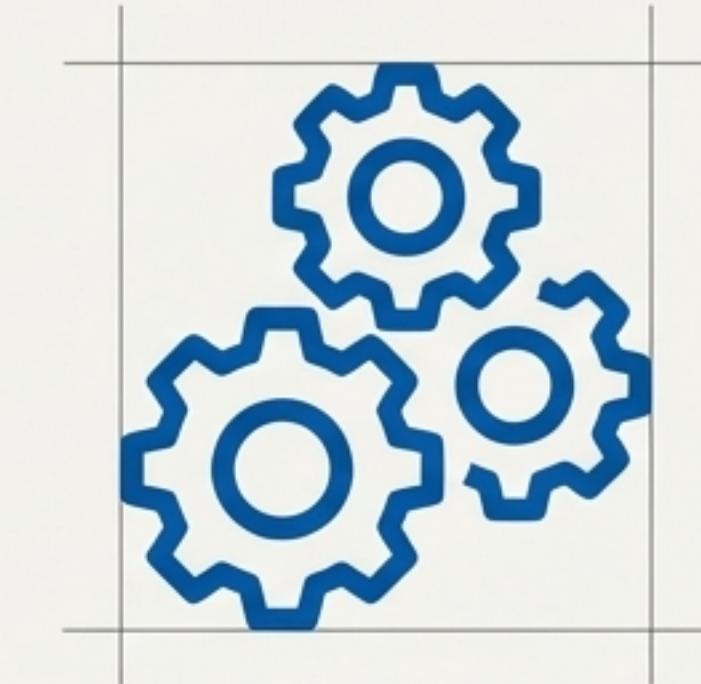
ランタイムのオーバーヘッドを排除したネイティブ実行速度と、「コード変更ゼロ」でハイエンドPCから安価なSBCまで展開できるスケーラビリティ。



## Architected Robustness

### アーキテクチャに根差す堅牢性

PLCの堅牢なデータモデルを構造的に強制。開発者のスキルに依存せず、フレームワークレベルでシステムの安定性を保証。



## Modern Development & Scalability

### 最新の開発手法と拡張性

標準C++とGit等のITツールをフル活用可能。複雑な要求に応える柔軟なモジュール構成と、将来の拡張を見据えた設計。

以下、これらの主張を技術的な事実で証明します。

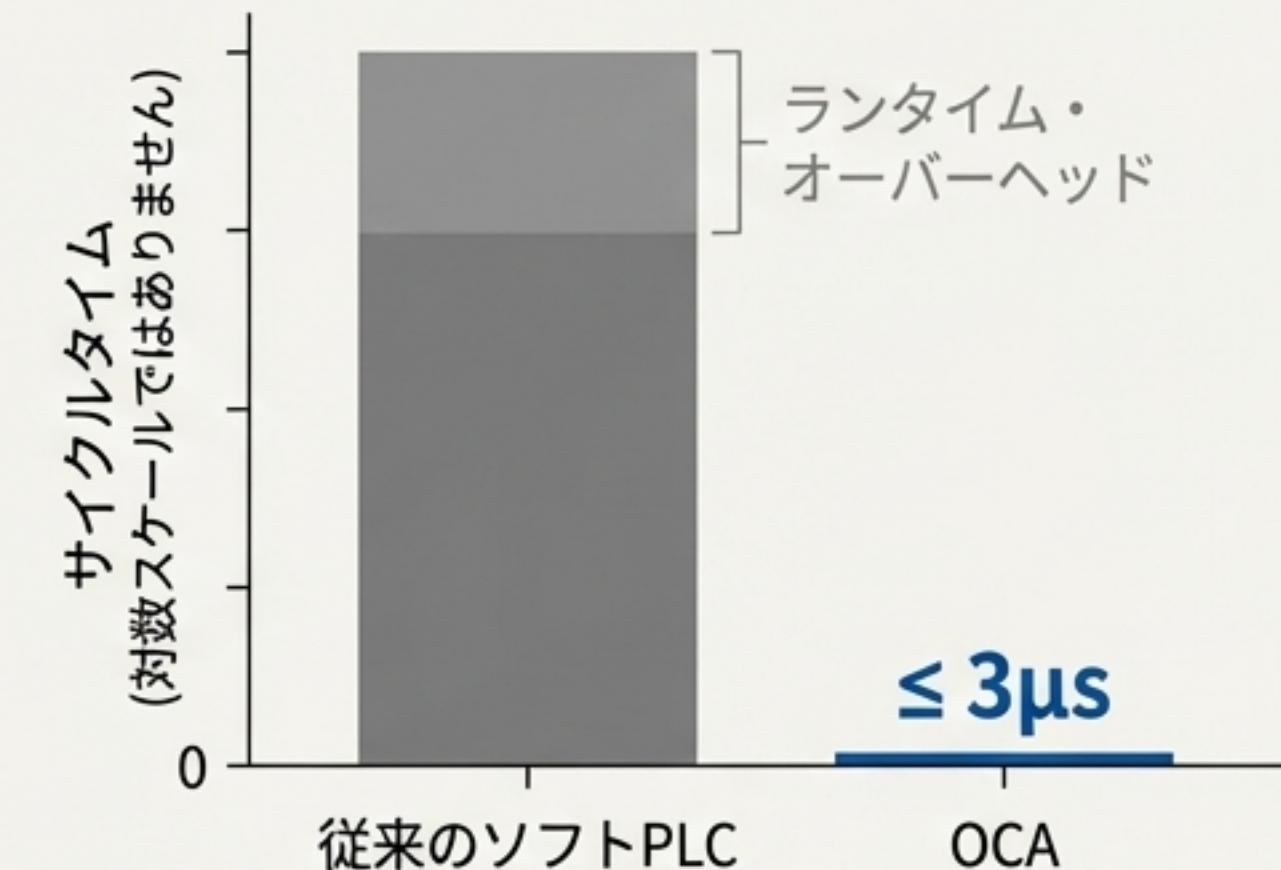
## 【証明①】汎用OSで実現する、マイクロ秒オーダーの実行性能

# 最速 3μs 以下

従来のソフトウェアPLCの常識を覆す、超最速オーダーの制御サイクル。

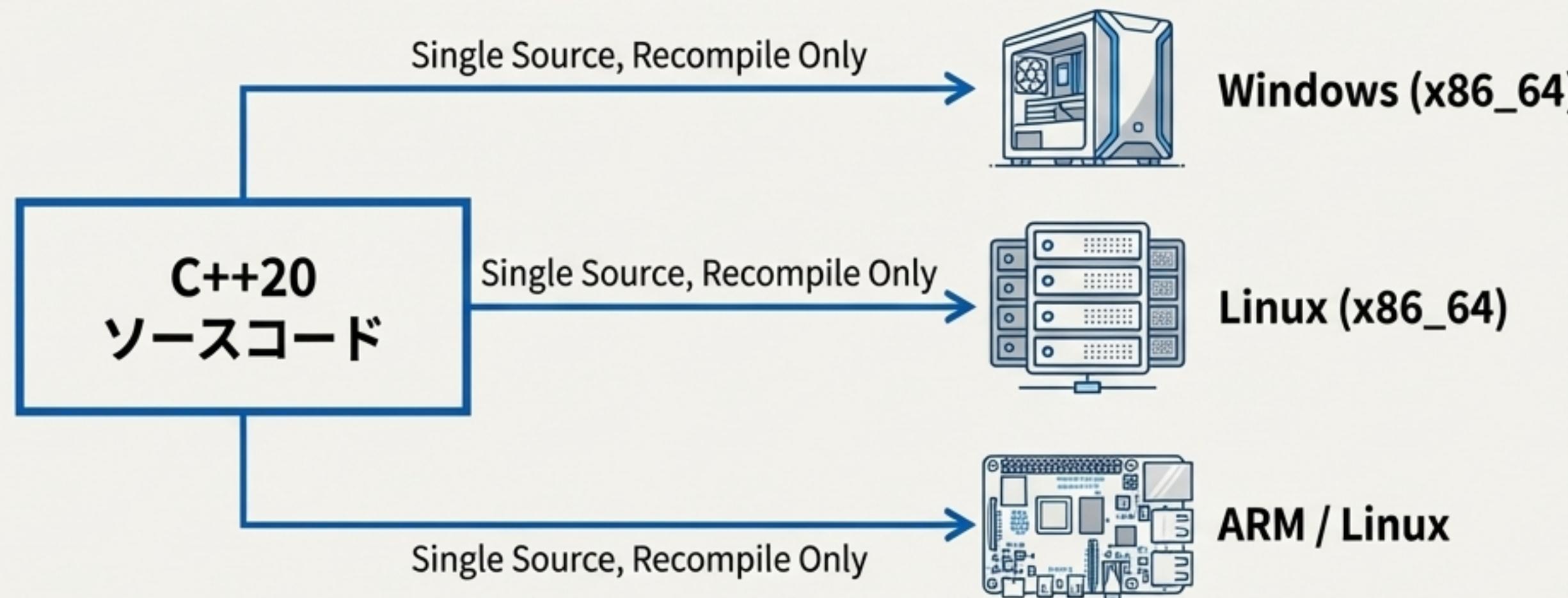
## なぜ実現できるのか？

- **ネイティブ実行**: CODESYSのようなランタイムに依存せず、OS上で直接コンピールされたバイナリとして動作。ハードウェアの性能を限界まで引き出す。
- **C++20による最適化**: 最新のC++20標準規格のみで記述。徹底したメモリレイアウトの最適化とオーバーヘッドの排除を設計段階から実施。
- **実証済みの性能**: ハイエンドなWindows PCだけでなく、わずか数千円の **Raspberry Pi Zero 2W** 上でも、専用機と互角の数マイクロ秒 ( $\mu\text{s}$ ) オーダーの制御を達成。



【証明①】ソースコードは一つ。ターゲットは無限。

## 「コード変更ゼロ」の真なるクロスプラットフォーム性



### アーキテクチャの優位性

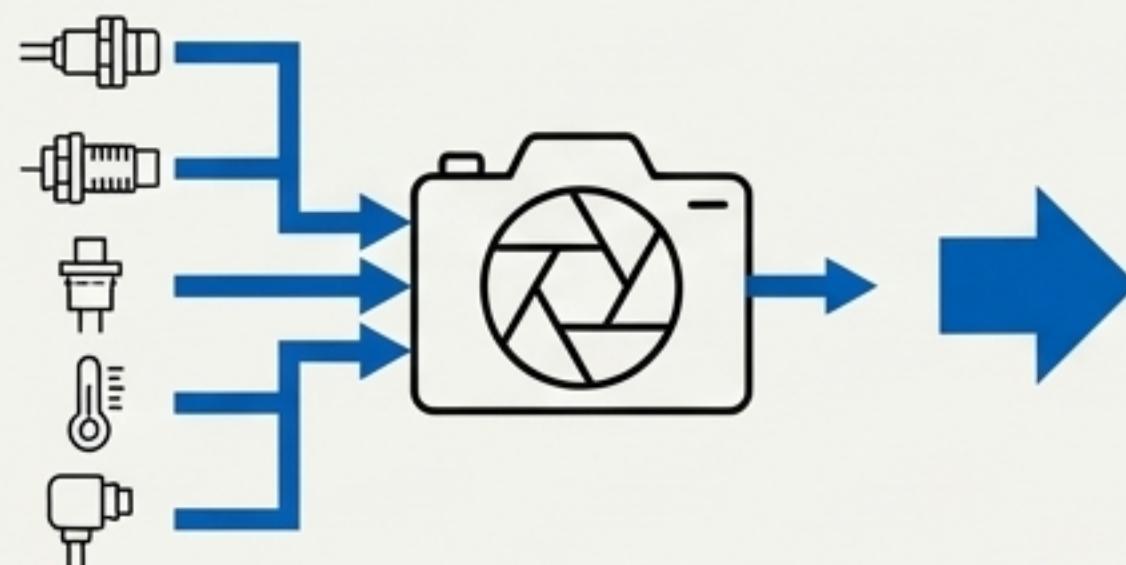
- コアエンジンは標準規格のC++20のみで記述。
- Windows固有の処理（DLL呼び出し等）やOS依存コードはアーキテクチャ内部で完全に抽象化・局所化。
- 開発者はOSやCPUアーキテクチャの違いを意識することなく、制御ロジックそのものに集中できる。

### ビジネスインパクト

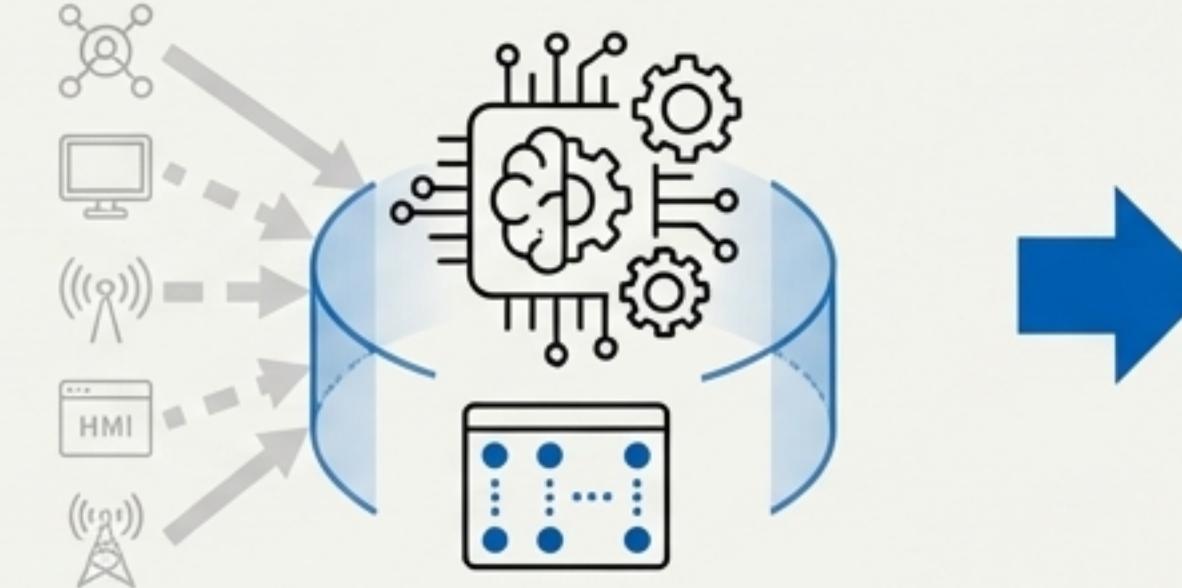
- ハイエンドな試作機から、コスト重視の量産機まで、全く同じソフトウェア資産をシームlessly展開可能。開発コストと期間を大幅に削減する。

## 【証明②】PLCの堅牢性の根幹を、アーキテクチャで強制する プロセスイメージによる絶対的なデータ整合性

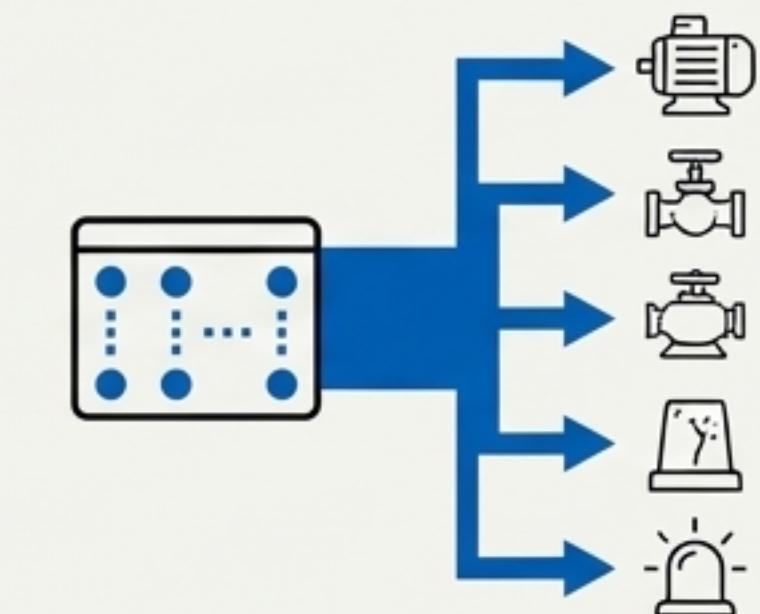
STEP 1: スナップショット入力



STEP 2: 決定論的ロジック実行



STEP 3: 一括出力



サイクル開始時に、**全入力データを一括で取得し**、その後の演算中は固定される。

演算中に外部（通信等）から入力データが変更されても、ロジックは影響を受けない。

全てのロジック演算完了後、**全出力データを一括で更新する**。

この「スキャン」単位での厳密なデータ管理により、PC制御で懸念される「演算中のデータ不整合」を構造的に排除。PLCと同様の堅牢なデータ整合性を保証する。

## 【証明②】リアルタイム性を損なわずに、データの安全性を守る協調的割り込み (Cooperative Interruption)

### 従来の強制割り込み (Preemptive)



データ競合 (Data Race) のリスク。

### OCAの協調的割り込み (Cooperative)



データの安全性を確保しつつ、システム全体の即応性を維持。

### 仕組み

優先度の高い要求が発生しても、実行中のロジックを強制的に中断しない。ロジックの区切り（ネットワーク境界）という安全なポイントで処理を受け入れる。これにより、データの破壊を防ぎ、決定論的な動作を維持する。

## 【証明②】 安全性を、開発者のスキルに依存させない 基底クラス‘NetworkBase’による構造的な安全性

### コード例（概念図）

```
// ユーザーはNetworkBaseを継承するだけ
class Bay1_ReadyLogic : public FaFramework::NetworkBase
{
public:
    // コンストラクタで入出力を宣言
    Bay1_ReadyLogic() {
        RegisterInput("B_Bay1_Status");
        RegisterOutput("LED_Bay1_Ready");
    }
protected:
    // 安全なコンテキストでロジックを記述
    void OnExecute() override {
        bool status = GetBool("B_Bay1_Status");
        // ...ロジック...
        SetBool("LED_Bay1_Ready", result);
    }
};
```

### フレームワークが自動で実行する安全機能

- ✓ **タグ品質チェック**: OnExecute() が呼ばれる前に、全入力タグの品質 (Quality) が Good であることを自動検証。通信異常時の誤動作を防止。
- ✓ **スレッドセーフなアクセス**: GetBool/SetBool などのアクセサが、スレッドセーフな排他処理を内包。開発者はロックを意識する必要がない。
- ✓ **例外ハンドリング**: ロジック内で発生した例外をフレームワークが捕捉し、システムの暴走を防ぐ。

開発者は「制御ロジックそのもの」に集中でき、システム全体としての堅牢性が担保される。

## 【証明③】IT業界の「当たり前」を、制御開発の現場へ

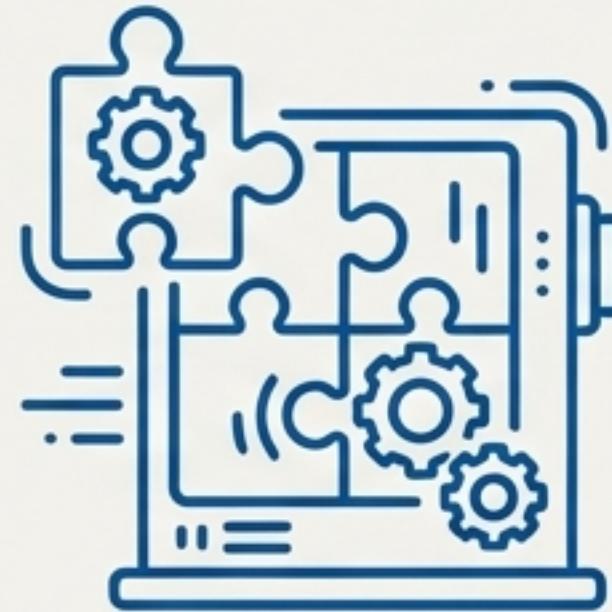


### OCAがもたらす開発プロセスの変革

- 標準ツールを活用: 特殊なPLC開発環境は不要。使い慣れたVisual StudioやVS Codeで開発が可能。
- 厳格なバージョン管理: Gitを用いて、誰が・いつ・何を・なぜ変更したのかを全て記録。変更の追跡や差し戻しが容易に。
- 品質の自動化: 単体テストフレームワーク (Google Test等) を導入し、ロジックの品質を自動で担保。
- CI/CDによるデプロイの高速化: ビルド、テスト、デプロイを自動化し、ソフトウェアの更新を迅速かつ安全に現場へ届ける。

「ラダーが書けるベテラン」への依存から脱却し、  
ITエンジニアを即戦力の制御エンジニアへと転換させる。

# 【証明③】現在の要求から未来の拡張まで、柔軟に応えるアーキテクチャ



## ① ハイブリッド・モジュール構成

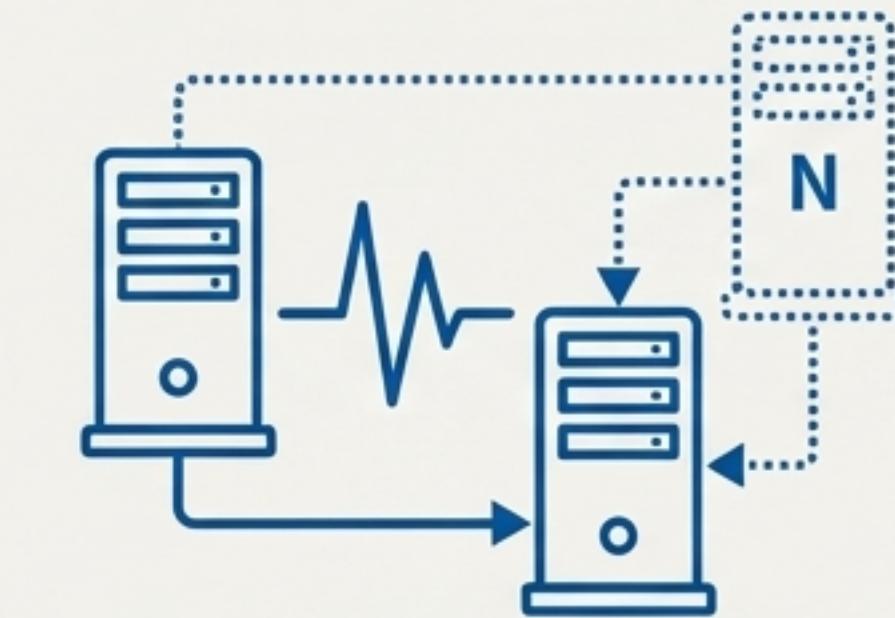
ロジックの実装形態を、要件に合わせて柔軟に選択可能。

- **動的インポート (DLL/so)** : 制御エンジンを停止することなく、ロジックの更新や機能追加が可能。
- **静的内包**: アプリケーション本体にロジックを組み込み、単一の堅牢な実行ファイルとして運用。



## ② 異種プロトコルの混在制御

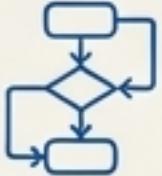
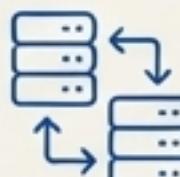
通信レイヤーが抽象化されており、物理層の異なるデバイスを統一的に扱える。S7通信 (Siemens) と EtherCAT といった異なるプロトコルを、同一プロジェクト内で混在させて制御可能。



## ③ N重化による高可用性

高価な専用ハードウェアは不要。ソフトウェアの標準機能として **N重化冗長化** をサポート。主系に障害が発生した際、即座に待機系が制御を引き継ぎ、システムのダウンタイムを極限まで抑制する。

# 技術的優位性のまとめ：CODESYSとのアーキテクチャ比較

比較項目	CODESYS	OCA
 アプローチ	PCをPLCとして振る舞わせる (ランタイム型)	PCのポテンシャルを制御に直結 (ネイティブ型)
 実行性能	ランタイムのオーバーヘッド あり。	<b>最速3μs以下。</b> ネイティブ実行 による超高速処理。
 クロスプラット フォーム	OSごとに専用ランタイムが 必要。	<b>単一ソースコード</b> でWindows, Linux, ARMへ展開可能。
 データ整合性	プログラマの設計に依存。 データ競合のリスクあり。	プロセスイメージを構造的に強制。 アーキテクチャが整合性を保証。
 開発手法	IEC 61131-3言語と専用 ツール。	<b>標準C++20。</b> Git, CI/CDなど 最新ITツールをフル活用。
 冗長化	専用ライセンスや特定 ハードウェアが必要。	<b>ソフトウェア標準機能。</b> 汎用ネットワークでN重化を実現。

# OCAは、次世代の産業用オートメーション開発基盤を定義する

OCAは、PC制御が抱えていた「信頼性への懸念」を、堅牢なアーキテクチャによって完全に払拭した。同時に、C++ネイティブによる「超高速・クロスプラットフォーム動作」と「最新のソフトウェア開発手法」を制御の世界に持ち込んだ。

これは、従来のPLCやソフトウェアPLCでは到達し得なかった、新しいオートメーションの形。OCAは、単なるソフトPLCの域を完全に脱し、「世界中のあらゆるハードウェアで動き、決して止まらない、次世代の産業用OS」へと進化する。

