JAIST Repository
https://dspace.jaist.ac.jp/

Title	IoTシステム開発の複雑さを低減するための統合的アーキテクチャ
Author(s)	栗林, 健太郎
Citation	
Issue Date	2025-03
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/19921
Rights	
Description	Supervisor: 篠田 陽一, 先端科学技術研究科, 博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

博士論文

IoT システム開発の複雑さを低減するための統合的アーキテクチャ

栗林 健太郎

主指導教員 篠田 陽一

北陸先端科学技術大学院大学
先端科学技術専攻
（情報科学）

令和 7 年 3 月

Abstract

This research proposes a Dynamic Isomorphic IoT System Architecture to reduce the complexity of development in IoT systems, which have traditionally been considered to have a heterogeneous configuration. In conventional IoT system development, different technologies and programming languages are often used in the device layer, edge layer, and cloud layer, leading to decreased development efficiency and interoperability issues due to this heterogeneity. Moreover, the complexity of coordination and data flow between layers increases, adding to the burden on developers.

To address these challenges, this research makes the following three proposals. First, we propose and implement a data flow platform that enables integrated design and development of the entire IoT system using a single programming language. Specifically, by leveraging the Elixir language and its ecosystem, we construct a consistent development environment from the device layer to the cloud layer. This eliminates technical gaps between layers and reduces the complexity of IoT system development. Second, we propose and implement a method that allows developers to dynamically apply code changes to applications within IoT devices. Traditionally, code changes to devices required firmware updates and reboots, delaying the development cycle. The proposed method allows immediate application of code changes to running devices, enabling a rapid development cycle and improving development efficiency. Third, we propose and implement a dynamic and isomorphic IoT system architecture utilizing WebAssembly (Wasm). Development using a single language may impose constraints on incorporating new technologies, and introducing dynamic update methods without guidelines may cause confusion in system design. The proposed method uses Wasm, which has portable characteristics, to easily incorporate necessary functionalities, using common Wasm binaries across layers and making those parts dynamically updatable, thereby balancing integration and flexibility.

To verify the effectiveness of these proposed methods, we conducted implementations and performance evaluations based on actual use cases. In the first proposal,

we developed a data flow platform named Pratipad and demonstrated that integrated design and implementation of data flows from the device layer to the cloud layer using the Elixir language is feasible. This enabled us to construct realistic IoT systems comprising multiple layers and provided a technical foundation that allows data flows to be described in an easily comprehensible manner, demonstrating its effectiveness on a realistic scale. In the second proposal, we proposed and implemented a method for dynamically applying code changes to IoT devices and experimentally showed that the time required for updates can be significantly reduced compared to existing firmware update methods. This accelerates the development cycle and reduces the developers' burden. In the third proposal, we evaluated the utility of an isomorphic IoT system using Wasm in use cases involving image recognition and machine learning models. By using common Wasm binaries across layers, we ensured functional consistency and reusability while enabling dynamic feature updates.

Overall, the proposed methods in this research address the challenges of heterogeneity in IoT system development, achieving both improved development efficiency and system flexibility, thereby reducing the complexity of IoT system development. This enables rapid development and deployment of IoT systems and allows for adaptation to future technological innovations, significantly contributing to the advancement of the IoT field.

Keywords: IoT system development; Dynamic isomorphic IoT systems; Elixir; Nerves; WebAssembly

概要

本研究は，異種混合的（heterogeneous）な構成が当然視されてきたiotシステムに対して，開発の複雑さを低減するために動的（dynamic）かつ同型（isomorphic）なiotシステムアーキテクチャ–Dynamic Isomorphic IoT System Architecture–を提案する．従来のIoTシステム開発では，デバイス層，エッジ層，クラウド層それぞれで異なる技術やプログラミング言語が用いられることが多く，その異種混合性から開発効率の低下や相互運用性の問題が生じていた．また，各層間の連携やデータフローの複雑性が増大し，開発者の負担が増加している．

本研究では，これらの課題を解決するために，以下の3つの提案を行う．第1に，IoTシステム全体を単一のプログラミング言語で統合的に設計・開発することを可能とするデータフロー基盤を提案・実装する．具体的には，Elixir言語とそのエコシステムを活用し，デバイス層からクラウド層まで一貫した開発環境を構築する．これにより，各層間の技術的なギャップを解消しIoTシステム開発の複雑さを低減する．第2に，IoTデバイス内のアプリケーションに対して，開発者がコードの変更を動的に適用できる手法を提案・実装する．従来は，デバイスへのコード変更にはファームウェアの更新や再起動が必要であり，開発サイクルが遅延していた．提案手法では，実行中のデバイスに対してコードの変更を即時に適用できるため，迅速な開発サイクルが実現し，開発効率が向上する．第3に，WebAssembly（Wasm）を活用した動的・同型なIoTシステムアーキテクチャを提案・実装する．単一の言語による開発では，新技術の取り込みに制約が生じる可能性がある．また，動的な更新方式を指針なく持ち込むことはシステムの設計に混乱をきたしかねない．提案手法では，ポータルブルな特性を有するWasmを用いて必要な機能を容易に取り込み，各層で共通のWasmバイナリを使用し，その部分を動的に更新可能にすることで，統合性と柔軟性を両立させる．

これらの提案手法の有効性を検証するために，実際のユースケースに基づく実装と性能評価を行った．第1の提案では，データフロー基盤としてPratipadを開発し，Elixir言語を用いてデバイス層からクラウド層まで一貫したデータフローの設計・実装が可能となることを示した．これにより，多階層から成される現実的なIoTシステムを構成可能で，データフローを見通しよく記述できる技術基盤を提案・実装し，また，それが現実的な規模で有効なものであることを示した．第

2 の提案では，IoT デバイスへのコード変更の動的適用手法を提案・実装し，既存のファームウェア更新方式と比較して，更新に要する時間を大幅に短縮できることを実験的に示した．これにより，開発サイクルが迅速化し，開発者の負担が軽減される．第 3 の提案では，画像認識や機械学習モデルを用いたユースケースにおいて，Wasm を活用した同型 IoT システムの有用性を評価した．各層で共通の Wasm バイナリを使用することで，機能の一貫性と再利用性を確保しつつ，動的な機能更新が可能であることを示した．

総合的に，本研究の提案手法は，IoT システム開発における異種混合性の課題を解決し，開発効率の向上とシステムの柔軟性を両立させることにより，IoT システム開発の複雑さを低減することができる．これにより，IoT システムの迅速な開発・展開や，将来的な技術革新への対応が可能となり，IoT 分野の発展に大きく貢献するものである．

キーワード: IoT システム開発，動的・同型 IoT システム，Elixir，Nerves，WebAssembly

目 次

第 1 章　序論 **1**

1.1　本研究の背景 1

1.1.1　IoT システムの異種混合性 1

1.1.2　統合的アーキテクチャへの取り組み 2

1.1.3　本研究のモチベーション 2

1.2　統合的アーキテクチャを実現する上での課題 3

1.2.1　課題 1：IoT システム全体を統合的に開発するための技術基盤 3

1.2.2　課題 2：IoT システムの柔軟性を向上するための動的に可変な性質 4

1.2.3　課題 3：新技術へ迅速に対応するための柔軟な構成 4

1.3　本研究の目的 5

1.3.1　IoT システムを統合的に開発可能なアーキテクチャ 5

1.3.2　技術的統合性の導入 6

1.3.3　動的な可変性の導入 7

1.3.4　構成的統合性の導入 8

1.4　本論文の構成 9

第 2 章　関連研究 **11**

2.1　課題 1 における関連研究 11

2.1.1　副課題 1.1：IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか 12

2.1.2　副課題 1.2：選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか 12

2.1.3　副課題 1.3：IoT システムの階層的なアーキテクチャにおけるデータフローを見通しよく扱えるか 14

2.1.4 課題 1 の解決方針 14
2.2 課題 2 における関連研究 15
2.2.1 IoT デバイスのソフトウェア更新 15
2.2.2 開発効率向上を目的とするソフトウェア更新方式 16
2.2.3 課題 2 の解決方針 17
2.3 課題 3 における関連研究 18
2.3.1 IoT システムに求められるデバイスの動的更新方式 18
2.3.2 WebAssembly を用いた IoT デバイスの動的更新手法 19
2.3.3 課題 3 の解決方針 20
2.4 本研究の位置づけ 21
2.4.1 Dynamic Isomorphic IoT System Architecture 21
2.4.2 提案するアーキテクチャの既存手法との比較 22

第 3 章 技術的統合性の導入 24

3.1 はじめに 24
3.2 提案手法 27
3.2.1 Elixir による技術的統合の実現 27
3.2.2 多様かつ双方向性をもつデータ取得方式への対応 30
3.2.3 可読性の高いデータフロー表現記法 32
3.2.4 提案手法を実装する上での技術的課題 38
3.3 評価 39
3.3.1 有効性の評価 39
3.3.2 適用可能性の評価 45
3.4 本章のまとめ 50

第 4 章 動的な可変性の導入 52

4.1 はじめに 52
4.2 提案手法と実装 56
4.2.1 提案手法 56
4.2.2 実装 57
4.3 実験と評価 60

4.3.1 対象と方法 60
4.3.2 実験結果 61
4.3.3 議論 63
4.4 本章のまとめ 64

第 5 章 構成的統合性の導入 65
5.1 はじめに 65
5.2 提案手法と実装 67
5.2.1 提案手法 68
5.2.2 実装 70
5.2.3 提案手法を実装する上での技術的課題 75
5.3 評価 76
5.3.1 動的 IoT アプリケーションのための Wasm 76
5.3.2 同型 IoT システムにおける Wasm 79
5.4 本章のまとめ 81

第 6 章 結論 83
6.1 本研究の成果 83
6.2 本研究の一般的な適用可能性 85
6.3 本研究の社会的な意義 86
6.4 今後の展望 86

図 目 次

1.1 本研究の目的とその実現のための筋道 5
1.2 提案するアーキテクチャの俯瞰図 6
1.3 本研究の俯瞰図 9

2.1 本研究の位置付け 22

3.1 push モードのシーケンス図 33
3.2 pull モードのシーケンス図 34
3.3 demand モードのシーケンス図 35
3.4 提案手法を用いたデータフローの記述例 38
3.5 提案手法を用いたデータ処理の記述例 39
3.6 提案手法を用いた IoT システムの構築例 41
3.7 実験に用いる push 方式のデータフロー定義 46
3.8 実験に用いる demand 方式のデータフロー定義 47
3.9 CPU 使用率の比較（10 プロセス） 49
3.10 CPU 使用率の比較（100 プロセス） 49
3.11 CPU 使用率の比較（1,000 プロセス） 49
3.12 メモリ使用率の比較（10 プロセス） 49
3.13 メモリ使用率の比較（100 プロセス） 49
3.14 メモリ使用率の比較（1,000 プロセス） 49

4.1 既存方式のシーケンス図 57
4.2 提案方式のシーケンス図 58

5.1 動的 IoT アプリケーションのための提案手法の概要 68
5.2 同型 IoT システムのための提案手法の概要 69

5.3 動的 IoT アプリケーションのための提案手法の実装 70
5.4 Wasmtube の動作のシーケンス図 71
5.5 Wasmtube がファイル更新を処理するシーケンス図 73
5.6 同型 IoT システムのための提案手法のユースケース 74
5.7 Elixir アプリケーションから Wasm への関数呼び出しの例 74

6.1 本研究の成果 83

表 目 次

2.1 課題 1 の関連研究に基づく解決方針の位置付け 14

3.1 IoT システムのデータフローを記述するために Pratipad が提供する記法 38
3.2 IoT システムの構築例に用いた構成 41
3.3 提案手法を用いた各データ取得方式に対応するデータフローの記述 44
3.4 提案手法を用いたエッジ層におけるプロセッサーの記述 44
3.5 実験環境 46
3.6 実験結果 47

4.1 IoT デバイスを構成するハードウェアおよびソフトウェア，それらに対応するプラットフォームの整理 54
4.2 方式（1）および（2）における更新のフェーズと計測方法 61
4.3 方式（3）における更新のフェーズと計測方法 61
4.4 実験環境 62
4.5 各方式によるコードの更新に要した時間の比較（単位：秒） 62
4.6 各方式における更新対象ファイルのサイズ 63

5.1 実験環境 76
5.2 性能比較実験の結果 77
5.3 実験環境 78
5.4 機械学習モデルを用いた性能比較実験の結果 79

第1章　序論

本研究は，異種混合的（heterogeneous）な構成が当然視されてきたIoTシステム開発に対して，IoTシステム開発の複雑さを低減することを目的として，動的（dynamic）かつ同型（isomorphic）なIoTシステムアーキテクチャ–Dynamic Isomorphic IoT System Architecture–を提案する．

1.1　本研究の背景

本節では，本研究の背景として，IoTシステムの異種混合性について述べた後，IoTシステム開発の複雑さを低減するための統合的なアーキテクチャの必要性について述べる．

1.1.1　IoTシステムの異種混合性

IoTシステムは，本質的に異種混合的であると考えられてきた [1,2]．この異種混合性は，デバイスの種類，データ形式，通信方法など，様々な要因からなる [3]．そのような多様な要素の統合は，IoTシステムの開発において大きな課題をもたらす [4,5]．IoTシステムを構成するネットワークの相互運用性を担保することが難しく [6]，また，IoTアプリケーションやデバイスの多様な性質が潜在的な脆弱性を生み出すため，セキュリティ上の懸念が生じる [7]．これらの課題にもかかわらず，IoTシステムの異種混合性は，さまざまな分野で幅広いアプリケーションやサービスを提供するために自然かつ不可欠なものと考えられている．

IoTシステムでは，システム全体をデバイス層，エッジ層，クラウド層のように階層的に構成するアーキテクチャが広く用いられている [8–10]．そのことが，IoTシステムの異種混合性に拍車をかけている．階層的アーキテクチャは，IoTシステ

ムにおけるレイテンシの低減，回復力の向上，およびローカルデータ処理を可能にする [11,12]．また，セキュリティの懸念に対処し，スケーラブルなシステムの構築を可能にする．一方で，このアーキテクチャの採用が IoT システムの相互運用性や異種混合性における問題を加速させる [13]．

1.1.2　統合的アーキテクチャへの取り組み

異なるプラットフォームやアーキテクチャ間で統合的に開発できる環境の実現は，コンピューティングの歴史において長年追求されてきた目標である．古くは 1960 年代にまで遡り，IBM の System/360 アーキテクチャは異なるハードウェア間での互換性を目指した先駆的な試みであった [14]．1970 年代には，UNIX オペレーティングシステムが C 言語で書かれ，異なるハードウェア上での移植性を大幅に向上させた．1995 年に登場した Java [15] は，“Write Once, Run Anywhere” というスローガンのもと，クロスプラットフォーム開発の新時代を切り開いた．Java の仮想マシン（JVM）は，異なるハードウェアやオペレーティングシステム上で一貫した実行環境を提供し，開発者の生産性向上と保守性の改善に大きく貢献した．この統合的な開発環境への取り組みは，現代のモバイルアプリケーション開発や Web アプリケーション開発においても継続されている．例えば，React Native [16] や Flutter [17] などのフレームワークは，単一のコードベースから複数のモバイルプラットフォーム向けアプリケーションを生成することを可能にしている．また，WebAssembly [18] は，ブラウザ上で高速に動作する低レベルの仮想マシンを提供し，様々なプログラミング言語からコンパイルされたコードを Web 上で実行することを可能にしている．

1.1.3　本研究のモチベーション

前述した，異なるプラットフォームやアーキテクチャ間で統合的に開発できる環境の実現へ向けての取り組みは，本研究が対象とする IoT システム開発の文脈においても，異種混合性がもたらす課題に対する潜在的な解決策を示唆している．すなわち，IoT システムにおいても，デバイス層からクラウド層まで一貫して開発・運用可能にすることで，IoT システム開発の複雑さの低減が期待できる見込み

がある．異なる階層間でのシームレスな連携が容易に可能になり，データの一貫性やリアルタイム性を強化できる．これにより迅速な意思決定や自動化が促進され，運用コストの削減につながることを期待できる．また，統合的なアーキテクチャは，スケーラビリティや柔軟性を高め，将来的な技術革新への適応力を向上させる．さらに，セキュリティ面においても，統一された技術を用いることによって脆弱性の早期発見と対応が可能となり，信頼性の高いシステム運用の実現につながる．総じて，統合的に IoT システムを開発可能なシステムアーキテクチャの実現は，IoT システムの性能向上と持続可能な発展を支える重要な要素となり得る．

1.2　統合的アーキテクチャを実現する上での課題

本節では，IoT システムを統合的に開発可能なアーキテクチャを実現する上での課題について，3 点に整理して述べる．

1.2.1　課題 1：IoT システム全体を統合的に開発するための技術基盤

IoT システム全体を統合的に開発するためには，デバイス層からクラウド層まで一貫して使用できる技術基盤が不可欠である．この技術基盤は，システムの各階層における機能要件や運用上の制約を包括的に満たす必要があり，それぞれの層に特有のニーズに対応可能な柔軟性と拡張性を備えていなければならない．しかしながら，IoT システムの各層に求められる機能や制約は多岐にわたっており，これらを単一の技術で包括的にカバーすることは困難である．物理デバイス層では低消費電力や小型化といったリソース制約が厳しく求められる一方で，クラウド層では大規模なデータ処理能力や高度なセキュリティ機能が必要とされる．このように，異なる層間での要求仕様が大きく異なる状況をカバーできる技術が必要となる．そのために求められる技術はふたつに大きく分けられる．すなわち，各層を実装するためのプログラミング言語とそれらの間の通信を実現するためのネットワークプロトコルとが必要になる．

1.2.2 課題 2：IoT システムの柔軟性を向上するための動的に可変な性質

統合的な IoT システムアーキテクチャーを実現する上で，システムを動的に変更可能な性質を持たせることが不可欠である．異種混合的なシステムは，各層が多様な技術を用いて構成されている特性上，変化する状況やニーズへの対応が容易であるという利点がある．一方で，統合的なアーキテクチャの実現により，かえってシステム全体の柔軟性を維持することが困難になることが考えられる．すなわち、異なる技術やプロトコルの統合にともない、モノリシックに構成された変化への対応力の低いアーキテクチャになりかねない．したがって，統合的なシステムアーキテクチャを効果的に実現しつつ，同時に，システムを状況やニーズへの変化に対して柔軟に対応させることが可能な手法が求められる．そのためには，将来的な拡張や技術革新にも対応できるよう，各層を構成するアプリケーションを動的に変更可能にする必要がある．IoT システムが動的な性質を有することで，システムは多様な要件や環境変化に柔軟に対応でき，開発プロセスの迅速化や適応性の向上が期待できる．

1.2.3 課題 3：新技術へ迅速に対応するための柔軟な構成

統合的な技術基盤を用いつつ，急速に進展する技術革新を柔軟に取り入れられる構成の導入が不可欠である．共通の技術基盤に基づく統合的なアーキテクチャを実現することは，システム全体の一貫性や開発・運用における効率性を向上させる一方で，新たな技術の採用が制約される可能性がある．昨今，例えば AI 分野における技術革新が著しく進展しているが，統合的な技術基盤を用いた IoT システムへのそれらの新技術の取り込みに際して，採用した技術が有する機能の不足等により，対応が困難になる可能性が考えられる．一方で，そのために新技術を導入する部分のみ別の技術基盤を用いることは，IoT システム全体の統合的な性質を損なう技術的複雑性を導入することとなり，そもそもの課題が再燃することになってしまう．そのため，統合性を維持しつつも，モジュール性や拡張性を備えた柔軟な構成を実現することが必要である．

1.3　本研究の目的

本研究の目的は，IoT システム開発の複雑さを低減することである．その達成のために，IoT システムを統合的に開発可能なアーキテクチャを提案する．本節では，その目的を実現するアーキテクチャ（後述の通り，これを Dynamic Isomorphic IoT System Architecture と名付ける）の概要について述べるとともに，それが備えるべき具体的な性質について検討する．

1.3.1　IoT システムを統合的に開発可能なアーキテクチャ

本研究の目的を実現する上で，前述の 3 つの課題のそれぞれに対応して，提案するアーキテクチャが備えるべき性質を以下の通り 3 つに整理して示す（**図 1.1**）．

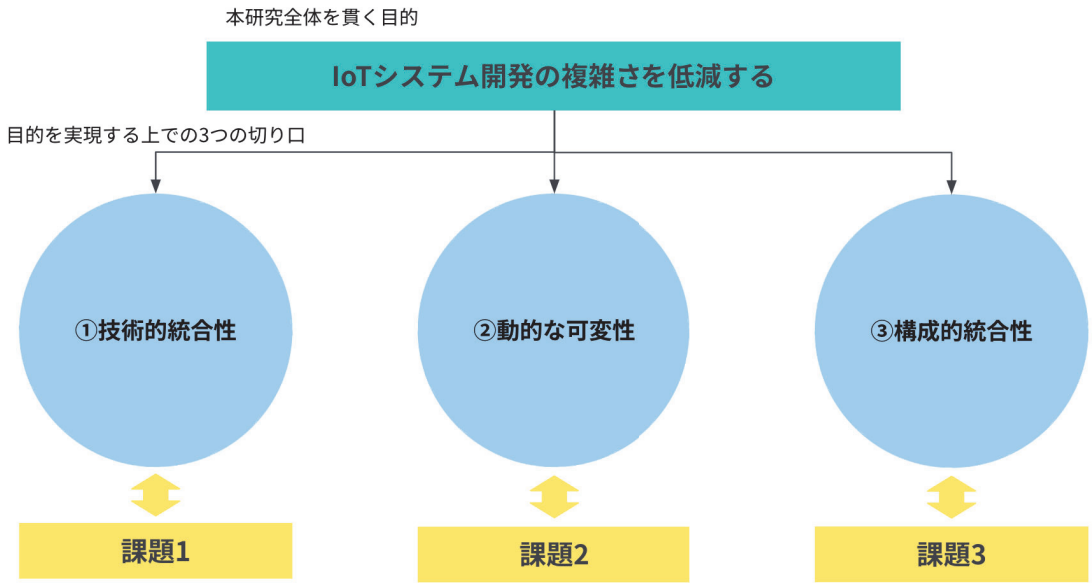

図 1.1: 本研究の目的とその実現のための筋道

これらの 3 つの性質を，IoT システムを統合的に開発可能なアーキテクチャとして実現するために階層およびその間の通信の構成にマッピングして示したのが**図 1.2** である．

この図は，IoT システムのデバイス層，エッジ層，クラウド層といった各階層がどのように相互に連携し，システム全体として機能するかを視覚的に表現している．図内の①から③は前述した 3 つの課題に対応した統合性の具体的な性質をそ

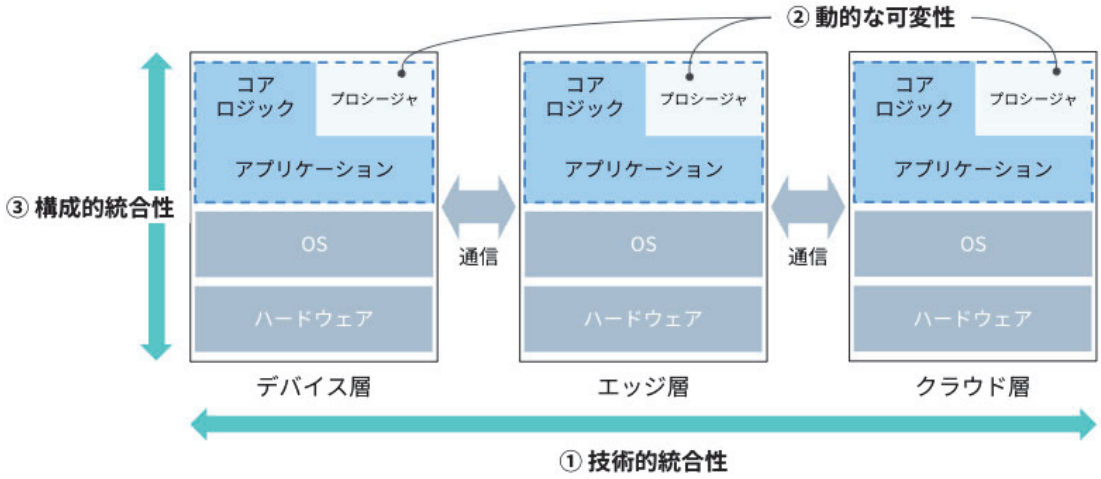

図 1.2: 提案するアーキテクチャの俯瞰図

れぞれ示している．①は「技術的統合性」を表しており，IoT システムの各層で使用される技術やプロトコルを統一することで，全体の一貫性と相互運用性を高めることを可能にする．②は「動的な可変性」を示し，システムが変化するニーズや環境に柔軟に適応できる能力を持つことを可能にする．③は「構成的統合性」を表し，新たな技術やモジュールをシステムに円滑に組み込むための柔軟な構成を可能にする．

本研究では，これらの統合性の具体的な実現方法について詳細に検討し，提案するアーキテクチャの設計と実装に反映させる．

1.3.2 技術的統合性の導入

まず，課題 1 として挙げた IoT システム全体を統合的に開発するための技術基盤を実現するために，**図 1.2** 内の「① 技術の統合性」を導入する．IoT システムの各階層を構成するアプリケーションを統合的に開発するために，それらを単一のプログラミング言語によって開発可能とする．さらに，それらのアプリケーションが各層間で統一された方法で通信できるよう，単一の通信プロトコルを用いる．

そのために，階層的なアーキテクチャからなる IoT システム全体を，単一のプログラミング言語で統合的に構築することを可能とする IoT システム開発基盤を提案する．その実現のために解決すべき課題として，(1) IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか，(2) 選択したプログラ

ミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか,(3) IoT システムの階層的なアーキテクチャにおけるデータフローを見通し良く扱えるか，の 3 点を示す．各課題に対して（1）各層の実装に用いるプログラミング言語として Elixir を選択する,(2）Elixir を用いて多様かつ双方向性をもつデータ取得方式に対応できる基盤として Pratipad を提案する,(3）Pratipad において階層的なアーキテクチャにおけるデータフローを一望のもとに把握できる記法を提供する，という 3 点の提案手法により解決を図る．提案手法について，有効性および適用可能性について評価する．その結果，提案手法が本研究の目的を実現するとともに，実用的な機能および規模を持つ IoT システムの構築に適用可能であることを示す.

1.3.3 動的な可変性の導入

次に，課題 2 として挙げた IoT システムの柔軟性を向上するために，**図 1.2** 内の「② 動的な可変性」を導入する．アプリケーション内のプロシージャを動的に変更可能にすることで，技術的な統合性を維持しつつ多様な要件や環境変化に迅速に対応でき，開発効率の向上が期待できる．ここで「動的（dynamic)」であるとは，IoT システムを構成するアプリケーションのコードをアプリケーション全体を再起動することなしに部分的に変更することで，その動作を変化させられることを意味する.

そのために，変化し続ける IoT システムへの多様な需要を満たすために，IoT デバイスの開発効率の向上を目的として，IoT デバイス内アプリケーションへの開発者によるコード変更の動的な適用手法を提案する．IoT デバイス内アプリケーションの開発において，開発者によるコードの変更を適用することで生じる動作の変更が意図した通りであるかどうかを確認するためには，変更内容をターゲットとなるデバイスへ適用し実際に動作させる必要がある．既存方式では，更新内容の生成および適用に加えて，デバイスの再起動に時間を要するため，迅速な開発サイクルの実現が困難である．本研究では，先行研究に基づきコードの変更をデバイスへ適用する方式について（1）ファームウェアイメージの全体を適用する方式,(2）ファームウェアイメージの差分を適用する方式,(3）アプリケーション

コードを動的に適用する方式の 3 つに分類する．その上で，開発効率の向上を目的として（3）を動的な性質を持つ言語によって実装し得る方式として位置づけ直して提案するとともに実装し，各方式について更新に要する時間を比較検討する．その結果，提案方式の既存方式に比べて更新に要する時間を評価し，目的が果たされていることを示す．

1.3.4 構成的統合性の導入

さらに，課題 3 として挙げた新技術へ迅速に対応するための柔軟な構成を実現するために，**図 1.2** 内の「③ 構成の統合性」を導入する．構成の統合性を，同型な（isomorphic）なアーキテクチャによって実現する．ここで「同型（isomorphic）」であるとは，IoT システムの各層においてアプリケーションの実行環境（使用するプログラミング言語やその実行基盤）として共通の構成を用いることで，各層を通じて同一のコードベースを用いてアプリケーションを構築および実行することを意味する．具体的には，IoT システムにおける各層のアプリケーションのコアロジックとプロシージャの構成を統合することで，特定のプロシージャのみにコアロジックとは異なる技術を用いた実装を取り込むことが可能になる．また，各層におけるその部分の実装が動的に可換になる．そのことで，技術的な統合性を維持しつつ動的な可変性を活かして，IoT システムに対して柔軟に新技術を導入することが可能となる．

そのために，WebAssembly（以下，Wasm と略記する）を使用して動的に更新可能な IoT デバイスを構築する手法を提案する．提案手法において IoT デバイスは，アプリケーションを実装する主要なプログラミング言語と Wasm ランタイムの組み合わせとして構成される．さらに，各システム層で共通のコードベースから構築された同一の Wasm バイナリを用いた機能を使用しつつ，その機能を部分的に更新可能な同型 IoT システムを提案する．具体的な使用例として，画像認識と分類のための機械学習モデルを Wasm バイナリにコンパイルし，IoT システムの各層を通じて同一の Wasm バイナリを使用して推論プロセスを実行する同型 IoT システムアーキテクチャを構築する．提案手法の有効性を確認するために定量評価を行う．提案手法は，アプリケーションから Wasm 関数を呼び出すことによる

オーバーヘッドを導入するが，その影響は限定的である．また，広く使用されている画像認識と分類モデルである ResNet-50 および MobileNetV2 を使用して提案手法の性能を測定する．提案手法は現状において実用的であり，将来に向けた可能性を提供することを示す．

1.4 本論文の構成

本節では，本論文の構成について述べる．

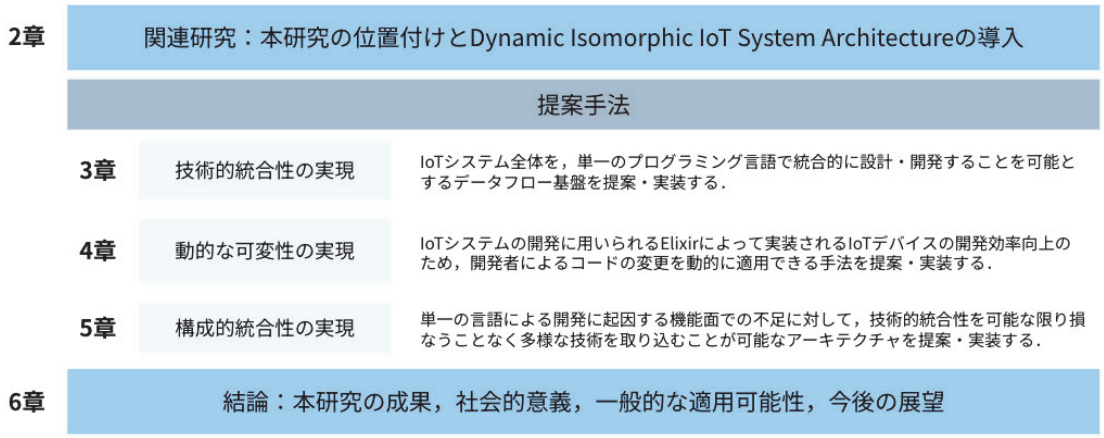

図 1.3: 本研究の俯瞰図

図 1.3 に本研究の俯瞰図を示す．まず，2 章で関連研究をまとめ，本研究を位置づけるとともに，本研究の目的である IoT システム開発の複雑さを低減するための，統合的な開発を可能とする Dynamic Isomorphic IoT System Architecture を導入する．次に，各提案手法について述べる．3 章では，IoT システムへの技術的統合性の導入について詳細に論じる．具体的には，IoT システム全体を単一の技術で統合的に開発可能とする基盤の提案とその実装手法を示す．この手法は，IoT システム開発における技術的統合性を実現するものであり，既報の [19] に基づいている．次に，4 章では，開発中の IoT デバイスへのコード変更を動的に適用する手法について詳述する．これにより，システムの停止や再起動を必要とせずに，変更内容を迅速に反映させることが可能となる．これは，IoT システム開発における動的な可変性を実現するものであり，既報の [20] に基づいている．さらに，5 章では，動的・同型 IoT システムアーキテクチャの提案とその有効性について考察す

る．このアーキテクチャは，各階層が同一の構造を持ち，動的な変更や拡張が可能である．これにより，IoT システム開発における構成的統合性を実現するものであり，既報の [21] に基づいている．最後に，6 章で，本研究の結論について述べる．

第2章　関連研究

本章では，前章で述べた課題について関連研究をまとめた上で，本研究の位置づけを示す．

2.1　課題1における関連研究

本節では，1.2.1 項で述べた課題 1 について，関連研究をまとめる．

課題 1 は，IoT システム全体を一貫して開発するための技術基盤をどのように構築できるかということであった．IoT システム全体を統合的に開発するには，デバイス層からクラウド層まで一貫して利用できる技術基盤が必要不可欠である．この技術基盤は，システムの各階層における機能要求や運用上の制約を全体的に満たす必要があり，各層特有のニーズに対応可能な柔軟性と拡張性を有していなければならない．

そのために，本節では課題 1 をさらに以下 3 つの副課題に分解して検討する．

1. 副課題 1.1：IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか
2. 副課題 1.2：選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか
3. 副課題 1.3：IoT システムの階層的なアーキテクチャにおけるデータフローを見通しよく扱えるか

2.1.1 副課題 1.1：IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか

副課題 1.1 を解決するためには，階層的なアーキテクチャからなる IoT システムにおいて，デバイス層，エッジ層，クラウド層のいずれをも単一の言語で開発できるプログラミング言語を選択する必要がある．また，それぞれの層を同一の言語によって開発できるだけでは不十分であり，各層の間を統合する通信プロトコルについても考慮されねばならない．さらに，その通信プロトコルは，副課題 1.2 の解決が可能である必要もある．

[22] は，IoT システムが複数の層にまたがることによって開発効率が低下するという課題を提示している．すなわち，それぞれの層を担当する専門性の異なる開発者同士が協力しあう必要のあることが，IoT システムの開発効率を損なうとする．当該研究は課題解決のために，独自に開発した言語を用いることで各層の詳細を知る必要なくシステム全体を統合的に開発できる TinyLink 2.0 を提案している．[23] もまた，同様の課題に対する Cross-site edge framework（CEF）と呼ばれる提案を行う研究である．CEF を用いると，各層の実装をプログラミング言語 Python を用いて，単一のファイルに記述できる．

2.1.2 副課題 1.2：選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか

副課題 1.2 を解決するためには，前項に述べた副課題 1.1 で選択したプログラミング言語を用いることで，以下にまとめるデータ取得方式に対応できることが必要となる．

[24] は，IoT デバイスからのデータの取得方式について，push 方式と pull 方式とがあるとする．push 方式では，デバイス層が起点となりクラウド層へデータを送信する．pull 方式では，デバイス層に対してデータの送信を要求することでクラウド層がデータを取得する．[25] は，push 方式と pull 方式とを組み合わせることで，取得要求に対して必要な分に限ったデータをデバイスから取得する方式を提案している．[26] は，データの流量がデータ処理パイプラインの許容量未満であ

る時に限ってデータ取得要求を送信するバックプレッシャーによる方式を提案している．処理が必要となるデータの流量を一定に抑えられるこの方式を，本研究では demand 方式と呼ぶこととする．

本研究が扱う IoT システムは，多階層から構成される．そのため，各層間のデータ通信がシステム全体の性能や効率に大きな影響を与える．特に，リアルタイム性が求められるユースケースや大量のデータを扱うシナリオにおいては，必要なデータを最大限取得・処理することが必要となり得る．例えば，産業用 IoT システムにおける設備の状態監視や，スマートシティにおける交通情報のリアルタイム分析などでは，データの流量を最適化しつつ高いレスポンス性を維持することが求められる．push 方式および pull 方式では，システム全体で処理可能なスループットをあらかじめ見込んだ上でデータの送信間隔を固定的に定めることになる．一方で，demand 方式を導入することで，IoT システムの過負荷を防ぎながら，必要な情報を適切なタイミングで取得することが可能となる．したがって，IoT システムにおけるデータ取得方式としては，push 方式および pull 方式に加えて，demand 方式も検討する必要がある．これにより，多階層の IoT システムにおけるデータフローを包括的に捉え，ユースケースに応じた柔軟なシステム設計を実現することが可能となる．

[24] は，データ取得方式として publish-subscribe 方式（以下，PubSub 方式）についても言及している．この方式は，デバイス層が特定のトピックに紐づくメッセージをブローカーへ送信すると，ブローカーを通じてそのトピックの購読者（subscriber）へメッセージが発行（publish）されて伝わる．前述の整理に基づくと，データフローの開始がどの層になるのかという点においては，PubSub 方式は push 方式と同様である．また，PubSub 方式を双方向に組み合わせることで pull 方式も実現可能である．その場合は，データフローはクラウド層から開始されることとなり，本項の整理における pull 方式同様となる．

また，クラウド層における分析結果に基づいてデバイスへの操作指示を行うためには，デバイス層からエッジ層を経由してクラウド層へ至る順方向のデータフローだけでなく，その逆方向のデータフローを行える双方向通信も必要となる．

表 2.1: 課題 1 の関連研究に基づく解決方針の位置付け

番号	副課題	関連研究による提案	解決方針の位置付け
1	IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか	TinyLink2.0 [22]，CEF [23] 副課題 1.1 のみなら有力な解決策だが，副課題 1.2，1.3 を解決できていない	3 つの副課題をすべて解決し，課題 1 に対する解決方針を実現できる手法を提案する
2	選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか	TinyLink2.0 [22]，CEF [23]，DDF/D-NR [29] push, pull, 双方向性を実現可能 (Websocket, Pub-Sub) ではあるが demand 方式に対応していない	
3	IoT システムの階層的なアーキテクチャにおけるデータフローを見通しよく扱えるか	Node-Red [28]，DDF/D-NR [29] ただし，Node-Red は各層を統合する実装は不可	

2.1.3 副課題 1.3：IoT システムの階層的なアーキテクチャにおけるデータフローを見通しよく扱えるか

プログラムをプリミティブな処理を行うノードが入出力を通じて連なる有向グラフとして表現するデータフローモデル [27] は，IoT システムにおいても複数の層を通じて行われるデータ通信を表現するために活用されている．一方で，階層的に構成された IoT システムのデータフローは，複雑なものとなり得る．そのため，データフローとその内部で行われるデータへの処理内容とを分離して記述することで，IoT システムの全体を通じたデータフローを見通しよく表現できる提案がなされている．Node-RED [28] は，IoT システムにおけるデータフローをビジュアルプログラミングによって見通しの良い形で実装するための，Web ブラウザ上で動作するアプリケーションである．GUI 上でノードを繋げていくことで，容易にデータフローを表現できる．また，[29] は Node-RED を拡張した Distributed Dataflow（DDF）を提案する．DDF では，Node-RED を拡張することでデバイス層とクラウド層を含むデータフロー全体を表現できる．

2.1.4 課題 1 の解決方針

本項で検討してきた関連研究は，3 つの副課題について部分的には解決策となり得るが，その全てを解決できるものはない．

TinyLink 2.0 と CEF は，単一の言語と通信プロトコルによって IoT システムを実装できるため副課題 1.1 の解決策として有力であるが，副課題 1.2 に関して整理

したデータ取得方式の全ては満たさない．また，データフローとデータへの処理内容とを分離して記述できないため，副課題 1.3 を解決しない．一方で，Node-RED とそれを用いた DDF は，ビジュアルプログラミングを用いたデータフロー記述の提案により，副課題 1.3 の解決策として有力である．しかし，Node-RED は 3 層を統合する実装が行えないため，副課題 1.1 を解決しない．また，DDF は 3 層を通じた実装を行えるが，通信プロトコルとして PubSub 方式に基づく MQTT を用いており demand 方式には対応していないため，副課題 1.2 を解決しない．

表 2.1 に，課題 1 と関連研究に基づく解決方針の位置づけをまとめた．IoT システム開発の複雑さを低減することを目的として，IoT システム全体を単一のプログラミング言語で統合的に構築することを実現するためには，課題を部分的にのみしか解決しない関連研究の取り組みでは，不十分である．本研究では，本章で検討してきた 3 つの副課題の全てを解決することで，課題 1 を解決できる手法を 3 章で提案する．

2.2 課題 2 における関連研究

本節では，1.2.2 項で述べた課題 2 について，関連研究をまとめる．

課題 2 は，IoT システムの柔軟性を向上させるための動的に可変な性質をいかにして実現できるかということであった．統合的なシステムアーキテクチャを効果的に実現しつつ，同時に，システムを状況やニーズへの変化に対して柔軟に対応させることが必要となる．本節においては，IoT システム開発の中でも特に IoT デバイスの開発効率の向上に焦点をあてて，開発者によるコードの変更をデバイスへ適用する方式について検討する．

2.2.1 IoT デバイスのソフトウェア更新

IoT デバイスは，一度配備された後も継続的にソフトウェアの更新が必要である．また，更新に際しては IoT デバイスのリソース制約の厳しさが課題となる．[30] は，IoT デバイスの製品ライフサイクルを初期・中期・後期の 3 段階に分類した上で，各段階において有効なセキュリティ対策についてまとめている．IoT デバイスのソ

フトウェア更新は製品ライフサイクルの全段階を通して必要であるとし，リソース制約の厳しいIoTデバイスに対していかにして効率的かつ安全に更新を実行できるかという観点で先行研究を検討している．[31]は，Wireless Sensor Networksを構成するデバイスへのソフトウェア更新について，広範なサーベイを行った．デバイスのデプロイ後のフェーズに着目し，フィールドに配置された，直接はアクセスできない多数のデバイスに対するソフトウェア更新を可能とする様々なプロトコルを検討している．[32]は，IoTデバイスのソフトウェア更新のプロセスを更新内容の検証，および，更新内容の配布の2つのフェーズに分類した上で，リソース制約の厳しいIoTデバイスに対してはエネルギー消費量のより少ない方式が有効であることを示している．

上記を背景に，開発者によるコードの変更をデバイスへ適用する方式について様々な提案がなされている．Ruckebuschらは，[33,34]においてコードの変更をデバイス上へ適用する方式を，書き換え対象に基づき（1）IoTアプリケーションの実装に用いられるスクリプト言語の特性を利用した動的なコード書き換え,（2）デバイス上で動作する仮想機械内で実行されるコードの書き換え,（3）デバイス上のファームウェアイメージの書き換え,（4）デバイス上のネイティブコードへの動的リンクの書き換えの4つに分類している．このうち（3）の方式は，更新対象となるファームウェアのサイズが他の方式と比較して大きいことから，ネットワーク経由での更新内容の転送に時間を要し，デバイスへの適用において電力をより多く消費する．そのため[35,36]は,（3）の方式による更新を効率化するために，ファームウェアイメージの生成プロセスをソースコードレベルでの類似性の向上，および，ファームウェアイメージの差分生成のアルゴリズムの改善の2つの観点で整理し，[35]はファームウェアイメージの差分のサイズを従来方式よりも縮小する方式を提案している．

2.2.2　開発効率向上を目的とするソフトウェア更新方式

本項では，IoTシステムの統合的開発を可能にしつつ，IoTデバイス内アプリケーションを柔軟に開発可能にするという観点から，開発者によるコードの変更をデバイスへ適用する方式を検討する．上記で検討した関連研究は，利用可能な

ネットワーク帯域，ハードウェアスペック，電源容量・確保等において制約の厳しいIoTデバイスに対して，いかにして安全かつ効率のよいソフトウェア更新が可能であるかを主たる課題としている．そのため，上述の開発者によるコードの変更をデバイスへ適用する方式を提案した [33,34] は，方式（1）および（2）について，それらの研究が対象とする強い制約の課されたデバイスにおいては実現困難であるとして，検討対象外としている．前述の通り [30] は，IoTデバイスのライフサイクル全体を通したソフトウェア更新が必要であることを述べている．また，コードの変更をデバイスへ適用する状況について [33] は，デバイスの配備前における開発・検証，および，デバイスの配備後における機能追加・更新・拡張の2つに分類しており，それぞれについて更新頻度を整理している．しかし，それらの関連研究はソフトウェア更新の主たる目的をデバイスの配備後におけるセキュリティ対策，バグ修正，機能拡張等に置いているため，IoTデバイス内アプリケーション開発効率の向上を目的とするソフトウェア更新の方式については検討されていない．そのため，本項の観点においては関連研究に加えてさらなる検討を要する．

2.2.3 課題2の解決方針

本項では，上述したIoTデバイス内アプリケーションの開発効率の向上という観点から，関連研究の提案した方式を整理し直す．関連研究における（1）から（4）の方式について，IoTデバイス内に配備されたコードを動的に変更可能であるという点に着目し，(1)（2）および（4）をひとつにまとめる．また，(3）に関する先行研究の積み重ねから，ファームウェアイメージによる更新については全体と差分による方式に分けることが妥当であると考える．その結果，IoTデバイス内アプリケーションへのード変更の方式を以下の3つに分類する．

1. ファームウェアイメージの全体を適用する方式
2. ファームウェアイメージの差分を適用する方式
3. アプリケーションコードを動的に適用する方式

[37] は，IoTデバイスの開発に用いられているボードを網羅的に総覧した上で，それらをスペックに応じてLow-end，Middle-end，High-endの3つに分類してい

る．製品化されたIoTデバイスの実装に際しては，デバイスの用いられる環境の制約やコストによってLow-endからMiddle-endのハードウェアが選ばれたとしても，開発環境としては，たとえばRaspberry Piのような High-endに分類されるハードウェアを用いることは可能である．また，IoTアプリケーションのプロトタイピング時や開発時においては，スクリプト言語や仮想機械上で動作する言語のような，ネイティブコードと比較してリソースを相対的に多く要求する言語を用いることは，開発効率の向上への寄与を目的として許容可能である．そのため，更新対象としてのIoTアプリケーションを構成するコードは，ネイティブコードはもとより，スクリプト言語や仮想機械上で動作する言語による動的な性質を持つコードであっても，開発時においては同等の機能を果たすとみなせる．よって，本節におけるIoTアプリケーションの開発効率の向上という観点からは，アプリケーションのコードの変更をデバイスへ動的に適用する方式を，上述の方式（3）にまとめて検討することが妥当である．本研究では，方式（3）に基づき課題2を解決できる手法を4章で提案する．

2.3　課題3における関連研究

本節では，1.2.3項で述べた課題3について，関連研究をまとめる．

課題3は，新技術へ迅速に対応するための柔軟な構成をいかにして実現するかということであった．統合的な技術基盤を用いつつ，IoTシステム開発における柔軟性を損なうことなく，急速に進展する技術革新を柔軟に取り入れられることが望ましい．そのためには，モジュール性や拡張性を備えた柔軟な構成を実現することが必要である．

2.3.1　IoTシステムに求められるデバイスの動的更新方式

新しいファームウェアイメージをデバイスに配布する方式（とりわけ無線経由での配布方式はOver The Airの頭文字をとってOTAアップデートと呼ばれる）が，IoTデバイスの更新に広く利用されてきた．しかし，この方式では一般に，ファームウェアイメージを適用するためにデバイスを再起動する必要があり，そのため

更新の完了に時間がかかる [32]．そのため，このアプローチでは，開発者がユーザーの要求に迅速かつ頻繁に対応することが困難である．この課題を解決するために，再起動を必要とせずに IoT デバイスを構成するコードを部分的に変更する手法についての研究が行われている．例えば，動的リンクを使用してコード参照を置き換えたり，スクリプトの動的な性質を利用して実行中のコードを更新したりする手法が提案されている [38]．また，本研究においては，課題 2 の解決方針で述べた通り，開発時のコード変更の適用を動的におこなえる手法を提案する．

一方で，クラウド・エッジコンピューティングにおける最近の発展を背景に，IoT デバイスにのみ焦点を当てるのではなく，エッジ層およびクラウド層を含む多層システムとして IoT システムを捉える必要がある [8]．したがって，多階層からなる IoT システムの効率的な開発と維持に適合する IoT デバイスの動的更新方法を確立することが必要である．また，課題 1 の解決方針に述べたとおり，本研究では IoT システムを統合的に開発可能な手法を提案する．一方で，統合的であることの反面において，システム開発が変化に対して硬直的になってしまっては課題 3 を解決し得ない．そこで，統合性をできるだけ損なわない形で，動的更新が可能でありつつ新技術の取り込みが容易な方式が必要となる．

2.3.2 WebAssembly を用いた IoT デバイスの動的更新手法

いくつかの先行研究が，Wasm [18] を用いて上述の問題を解決しようと試みている．Wasm は元々，Web ブラウザ上での処理を高速化するために開発された，スタックベースの仮想マシン（VM）の命令を記述するためのバイナリ形式である．現在，Wasm はその移植性の高さおよび安全な仕様と実行環境により，Web ブラウザ外でも利用されている．[39] は，再起動を必要とせずに IoT デバイスを動的に更新する手段として Wasm を利用できる可能性について示唆している．[40] は，Wasm ランタイムを IoT デバイスに軽量なコンテナとして組み込み，その上でアプリケーションを実行することで，IoT デバイスの動的な更新が可能であることを提案した．[40] は，IoT デバイスの実装に広く用いられているマイクロコントローラの ESP32 [41] 上で動作するアプリケーションに Wasm ランタイムを組み込み，その上で動作する Wasm バイナリを動的に書き換えることでデバイスの挙動を動

的に変更する実装を実証した．この実装では， Wasm ランタイム上で動作する関数との引数のやり取りは，限られたプリミティブ，整数，および浮動小数点数のみをサポートしており，デバイスとの限定的な相互作用のみを提供している．また，Wasm バイナリを更新する一般的な手法は提供されていない．

2.3.3 課題 3 の解決方針

統合的な開発基盤に対して Wasm を適用する上では，統一的な指針が必要である．そうでなければ，システムが場当たり的なパッチワークになりかねず，結果的に統合性を毀損してしまいかねないからである．[42] は，IoT システムの複数の層を通じて共通のコードベースを使用する同型（isomorphic）IoT システムの概念を提案している．クラウド／エッジコンピューティングの進展にともない，現在の IoT システムは IoT デバイスだけでなく，異なるプラットフォーム，アーキテクチャ，ハードウェアで構成された多階層からなるシステムとして成されている．そのため，IoT システムは自然と複雑なものとなり，開発者は各層を構成する様々な技術に精通する必要がある [43]．そこで，IoT システムの各層を実装するために共通の構成とコードベースを使用することで，新技術の容易な取り込みを可能にしつつ統合性を維持できることを期待できる．一例として，[42, 43] は先述した Wasm の移植容易性に焦点を当て，IoT システム全体でのアプリケーション構築における Wasm の使用を提案している．しかし，これらの研究で提案された同型性は，各層が同じ方法で Wasm ランタイムを使用している点に制限されており，同じ Wasm バイナリを使用してアプリケーションを構築および実行することには拡張されていない．さらに，これらの研究の提案は概念的な水準での提案にとどまっており，具体的な実装を提供していない．5 章では，実際のユースケースに基づいた Wasm バイナリを実行するレベルで動的・同型な IoT システムを構築する方法の実装について述べる．

2.4 本研究の位置づけ

本節では，上記で述べた課題とそれらの解決方針を統合する本研究の位置づけについて述べる．

2.4.1 Dynamic Isomorphic IoT System Architecture

上述した 3 つの課題に対する解決方針をまとめることで成立する IoT アーキテクチャを，本研究では Dynamic Isomorphic IoT System Architecture（動的・同型な IoT システムアーキテクチャ）と名付ける．本研究では，技術的統合性に加えて，動的な可変性による変化への柔軟性を持ちながらも，構成的統合性を維持できる IoT システムアーキテクチャを目指す．ここで「動的（dynamic）」であるとは，IoT システムを構成するアプリケーションのコードをアプリケーション全体を再起動することなしに部分的に変更することで，その動作を変化させられることを意味する．「同型（isomorphic）」であるとは，IoT システムの各層においてアプリケーションの実行環境（使用するプログラミング言語やその実行基盤）として共通の構成を用いることで，各層を通じて同一のコードベースを用いてアプリケーションを構築および実行することを意味する．

本章の各節で，関連研究の整理に基づいて 3 つの課題に対してそれぞれどのような方針で解決を図るかについて述べた．それらの解決方針をまとめた内容を，本研究の位置づけとして図 2.1 に示す．

2.1 節では，課題 1 に対して，IoT システムの開発において統合的な開発基盤を実現するために，IoT システムに対して技術的統合性を導入する解決方針を示した．一方で，統合的な技術基盤の導入により，IoT システム開発における柔軟性が損なわれてしまう可能性がある．すなわち，システム全体がモノリシックに構成されることによって状況の変化への対応がかえって困難になりかねないということである．その懸念が課題 2 を惹起する．

そこで 2.2 節では，技術的統合性と柔軟性を両立させるために，IoT デバイス内アプリケーションの開発において動的な可変性を導入する解決方針を示した．IoT システムを構成するアプリケーションに対して動的な構成の導入により柔軟性は担保されるものの，指針を欠いた柔軟性の導入は IoT システム全体としての統合

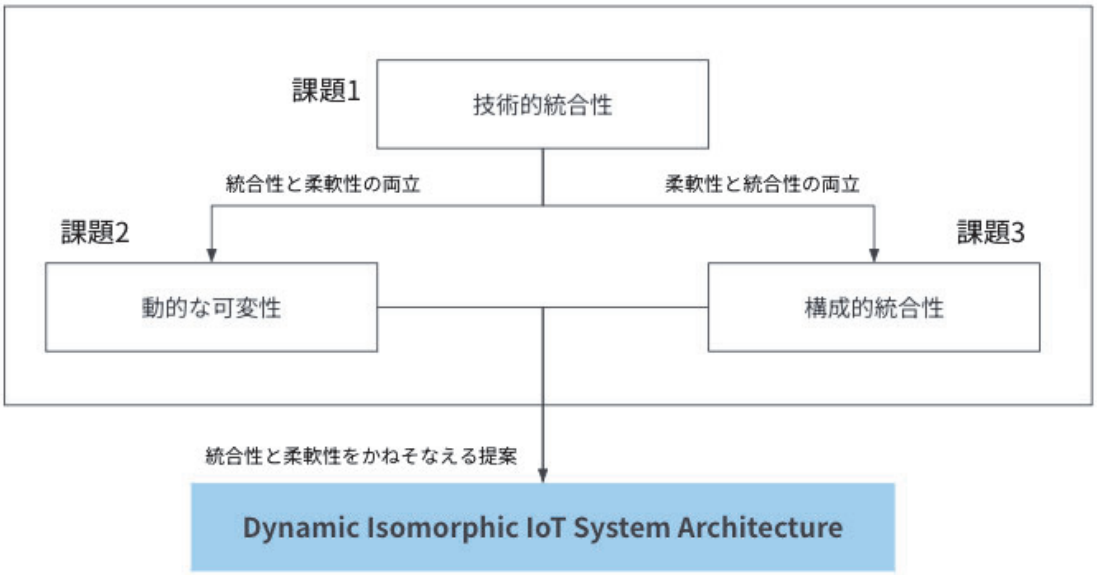

図 2.1: 本研究の位置付け

性を損なってしまう可能性がある．一方で，技術的統合性に拘泥してしまうと，採用した技術要素外からの新技術を取り込むことが困難になってしまう．その懸念が，さらに課題 3 を惹起する．

そこで 2.3 節では，IoT システムの開発において柔軟性と統合性とを両立するために，構成的統合性を導入する解決方針を示した．このようにして，3 つの課題のすべてについて同時に解決するための，統合性と柔軟性とを兼ね備える解決方針として，本研究が提案する Dynamic Isomorphic IoT System Architecture を位置付ける．

2.4.2 提案するアーキテクチャの既存手法との比較

本研究が提案する Dynamic Isomorphic IoT System Architecture は，デバイス層，エッジ層，クラウド層にわたる全体的な統一モデルに基づいて IoT システムを構築・運用するための枠組みであり，前述した 3 つの課題に対する解決方針を統合するものである．従来の IoT システム開発では，エッジ層やデバイス層において様々な要件に対して個別に対応する実行環境や手続きに依存せざるを得ず，その結果として IoT システムは異種混合的に構成されることになってきた．一方で，IoT システムを階層間を通じて開発可能にするプラットフォームも出てきている

が，クラウド層でのスケーラブルな処理やサービス活用が重視される一方で，後述の通り特定のベンダーや基盤への依存性が高まるというデメリットがある．

AWS Lambda [44] はクラウド上での FaaS（Functions as a Service）基盤としてスケーラブルな処理を容易にするとともに，AWS IoT Core [45] や AWS IoT Greengrass [46] を組み合わせることでエッジ側へ機能を拡張できる．しかし，これらは本質的に AWS のクラウドサービス群への依存を前提としており，全層で同一の概念的フレームワークを適用して実行中のロジックを即時に組み替えるような動的な可変性にも限界がある．Azure IoT Edge [47] はクラウド上の設定やポリシーに基づいてエッジデバイスにコンテナ化されたワークロードを展開し，クラウド層とエッジ層間の機能的連携や拡張を容易にする．しかし，エッジ層はあくまでクラウド層からの指示によりコンテナを実行する対象であり，IoT システムの全層が同一の抽象モデル上で統合されるわけではない．また，再構成にはクラウド層主導の再デプロイが前提であり，実行中の論理的な切り替えを直接的かつ柔軟に行うことは困難である．KubeEdge [48] は Kubernetes 基盤を拡張することでクラウド層とエッジ層間を統合し，Kubernetes による運用管理手法やスケーリングメカニズムをエッジ層へ拡張する．しかしながら，その内部モデルは Kubernetes リソースに依存しており，デバイス層を同列の概念モデル下で扱うことは想定されていない．さらに，Kubernetes 環境への依存は残り，特定のオーケストレーションモデルに制約される．

これらの制約に対し，Dynamic Isomorphic IoT System Architecture は IoT システムの全層で同型的にアプリケーションを構成し，また，アプリケーションの動作に対する動的な可変性を備えることで，多層間を統合的に開発することを可能にしつつ，特定基盤へのロックインや非動的な運用フローなど，従来のプラットフォームが提供するアーキテクチャが抱えてきた問題を克服する新たな方向性を示す．これにより，本研究の目的である IoT システムに対して，より一般的な手法として，開発の複雑さを低減するための統合的なアーキテクチャの実現が可能となる．次章以降，3 章–5 章を通じて，提案手法の具体的な実現方法および有効性について詳述する．

第3章　技術的統合性の導入

本章では，IoT システムの単一技術による統合的開発基盤を提案・実装することで，IoT システムの開発に対して技術的統合性を導入する．

3.1　はじめに

物理空間上のセンサーやアクチュエーター等のデバイスとサイバー空間上の計算処理とを架橋する IoT システムは，様々な領域において適用事例をもたらしている．[49] は，本章が対象とするクラウドコンピューティングをベースとした IoT システムの適用例として，スマートホームのような家庭での利用はもとより，医療用途やロジスティクス等の産業用途，エネルギーやモビリティに関わる社会インフラ基盤にいたるまで，多岐にわたる領域を挙げている．IoT システムでは，デバイスによってセンシングされた物理空間上のデータが，ネットワークを通じてサイバー空間上のシステムへと送信される．そして，集められたデータを用いてサイバー空間上で分析した結果に基づき，物理空間上のデバイスへのアクチュエーション指示が行われる．そのため，物理空間とサイバー空間との間における双方向のデータフローの構成が重要な課題となる．そのようなデータフローを構成するために，IoT システム全体を階層的に構成するアーキテクチャが広く用いられている [8, 10]．

本章においては，各階層を構成する計算資源の物理的な配置に基づき，(1) デバイス層，(2) エッジ層，(3) クラウド層の 3 層において構成される IoT システムのアーキテクチャーモデルを検討する．階層的なアーキテクチャを採ることにより，各層に専門特化した分業が行える．また，分業体制においてはそれぞれ各層の特性に則した多様な技術要素の選択，組み合わせにより開発を行える．一方で，[22] が指摘する通り，アーキテクチャの階層性をそのまま IoT システムを開発する組

織の境界と同一視してしまうことにより，それぞれ異なる技術によって構成される階層間をまたいで開発者たちが協働する必要があり，開発速度が低下してしまう問題が生じるという欠点についても考慮する必要がある．

本章は，階層的なアーキテクチャからなる IoT システム全体を，単一のプログラミング言語で統合的に構築することを可能とするデータフロー基盤を提案する．

大規模な IoT システムの開発には通常，階層的なアーキテクチャに基づく分業により多様な関係者がたずさわるが，デバイス層，エッジ層，クラウド層の各層を通じてシステム全体を単一の組織で開発することも可能である．例えばスマートリモコンの Nature Remo[1]では，デバイス層からクラウド層まで単一の組織において開発が行われている [50]．

単一の組織による開発を前提とすると，用いられる技術要素，特にプログラミング言語について単一のものを用いることができれば，学習コストと開発チーム間でのコミュニケーションコストが低減し，開発効率の向上が望める．また，技術要素が多ければ多いほど，サプライチェーンの保証が困難になることによるセキュリティリスクが発生する．このリスクを軽減するためにも，技術要素はできるだけ少ないことが望ましい [51]．よって，単一の組織によって IoT システム全体を開発するという前提においては，利用するプログラミング言語もまた単一のものを使える基盤を提案することに有用性を認められる．

本章の目的を実現するには，IoT システムの全ての層を実装可能なプログラミング言語を選択する必要がある．さらに，単一の技術要素による前述した恩恵を享受するためには，単に各層において同一の言語を使うだけでなく，IoT システム全体を統合的に構築可能である必要がある．そのためには，[29] の述べる通り，階層的なアーキテクチャからなる IoT システムのデータフローにおいて階層間をまたいで実現されるパーセプション–アクションサイクルに対応する必要がある．また，各層に点在してしまいかねないデータフローの記述を統合的に扱える必要もある．ここまでを，以下の 3 つの課題としてまとめる．

1. IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか．

[1] https://nature.global/nature-remo/

2. 選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか.
3. IoT システムの階層的なアーキテクチャにおけるデータフローを見通しよく扱えるか.

課題（1）を解決するためには，各層を構成するソフトウェアの設計・実装に用いられ得るプログラミング言語の多様な選択肢から，IoT システムを統合的に構築可能で，かつ，他の 2 つの課題を解決可能な言語を選択する必要がある.

課題（2）を解決するためには，課題（1）で選択したプログラミング言語によって，デバイス層からの多様なデータ取得方式や双方向の通信に対応できる基盤を構築可能である必要がある．ここでいう多様さとは，データ取得方式に次の複数の種別が存在することを指す．すなわち，階層的なアーキテクチャからなる IoT システムのデータフローにおけるデータ取得方式には push 方式と pull 方式 [24]，および，demand 方式 [25,26] があることを示す．push 方式では，デバイス層が起点となりクラウド層へデータを送信する．pull 方式では，デバイス層に対してデータの送信を要求することでクラウド層がデータを取得する．demand 方式では，エッジ層から必要とする分だけのデータをデバイス層へ要求することでデータを取得する．また，ここでいう双方向性とは，デバイス層からエッジ層を通じてデータが送信されるだけでなく，クラウド層からデバイス層への通信も発生し得ることを指す．典型的な例として，クラウド層における分析結果に基づいてデバイスへの操作指示を行うためには，双方向の通信が必要となることが挙げられる.

課題（3）を解決するためには，課題（2）に加えてエッジ層におけるデータ処理の記述も加わるため，課題（1）で選択したプログラミング言語がデータフローの全体像を見通しよく記述できる基盤を構築可能である必要がある.

本章では，IoT システム全体を単一のプログラミング言語で統合的に構築することを可能とするデータフロー基盤を実現するために，上述した 3 つの課題にそれぞれ対応する以下の解決策を提案する.

1. 各層の実装に用いるプログラミング言語として Elixir を選択する.
2. Elixir を用いて多様かつ双方向性をもつデータ取得方式に対応できる基盤として Pratipad を提案する.

3. Pratipad は階層的なアーキテクチャにおけるデータフローを一望のもとに把握できる記法を提供する．

提案（1）で，プログラミング言語 Elixir [52] を選択するのは，それがデバイス層，エッジ層，クラウド層のいずれをも開発可能な言語だからである．デバイス層の実装には，Elixir による IoT デバイス開発基盤である Nerves [53] を用いる．エッジ層の実装には，筆者の開発したデータフロー基盤である Pratipad [54][2]を用いる．クラウド層の実装には，Elixir による Web アプリケーションフレームワークである Phoenix [56] を用いる．また，各層は Erlang/OTP [57] の分散ネットワーク上に構築し，Pratipad が定めるプロトコルを用いて通信する．

提案（2）については，提案（1）で選択した Elixir を用いて，階層的なアーキテクチャからなる IoT システムにおけるデータフローを実現する基盤である Pratipad を開発した．Pratipad は，分散 Erlang ネットワークプロトコルと Elixir によるデータ処理基盤である Broadway [26] を組み合わせることで，課題（2）に挙げられているどのデータ取得方式にも対応し，双方向通信も可能なデータフロー基盤である．

提案（3）については，階層的なアーキテクチャからなる IoT システムにおけるデータフロー全体を表現可能な記法を，Pratipad に実装した．Pratipad では，デバイス層，エッジ層，クラウド層を一気通貫する双方向のデータ入出力と処理の流れを，同一ファイル内で宣言的に記述できる．

3.2 提案手法

3.2.1 Elixir による技術的統合の実現

課題（1）は，IoT システム全体を統合的に構築可能なプログラミング言語としてどの言語を選択するかということであった．本節ではプログラミング言語 Elixir を選択することで，Elixir を中核として各層を統合的に設計・実装できる手法を提案する．

[2]Pratipad について，筆頭による ElixirConf 2021 での発表資料 [55] でも詳細に説明されている．

プログラミング言語に関する選択肢の検討

Elixir と他の広く用いられているプログラミング言語（Python，C/C++，JavaScript，Go）を対象に，IoT システムを統合的に設計・開発する言語としてどれがふさわしいかという観点から，階層を通じて開発可能な包括性，対応する通信プロトコルの 2 軸において比較する．

次目で詳述する Elixir は，IoT デバイス開発プラットフォームや Web フレームワークを有することで， IoT デバイスからクラウドまでを開発可能な環境を提供する．Python は，MicroPython [58] という IoT デバイス開発に適した実装があり，また Python 自体が豊富なライブラリを持ち幅広い領域に適用可能である．C/C++ は，特にデバイス層での実装において広く活用されてきた実績がある，JavaScript にも Espruino [59] のような IoT デバイス向けの実装があり，Web 領域で確立されたノウハウを IoT システム全体の開発に適用可能である．クラウド層の実装に用いられることの多い Go も，TinyGo [60] という組み込みデバイス向けの実装を有しており，デバイス層の開発においても選択肢となり得る．

開発環境の包括性という観点においては，上述で検討したどの言語においても大きな差異は存在しない．一方で，通信プロトコルへの対応については，Elixir が一線を画している．MQTT などの IoT システムに適した通信プロトコルは，どの言語においてもライブラリによる実装があるため利用可能である．それとは異なり Elixir は，その実行基盤である Erlang/OTP による組み込みの分散ネットワークプロトコルを有している．当該プロトコルの利用を通じて，プロセス間メッセージパッシングによる複数ノード間の透過的通信，故障検出，動的なノード追加・削除等が標準機能として活用できる．この特性により，本研究が扱う多階層からなる IoT システムにおける統合的な開発が可能となる．したがって，他言語が外部ライブラリやプロトコルスタックの組み合わせを必要とする中，Elixir は言語ランタイムそのものを通じて分散通信と可用性を確立し，IoT 開発基盤として優位性を示していると考える．

Elixir の特徴と提案手法における活用

上述の通り，各層を統合的に設計・実装するのに適したプログラミング言語として Elixir を選択し，中核とする手法を提案する．通信プロトコルとしては Elixir の基盤となる Erlang/OTP の，TCP による分散ネットワークプロトコル [61] を用いる．デバイス層の実装には，Elixir による IoT デバイス開発基盤である Nerves を用いる．エッジ層の実装には，筆者らが Elixir を用いて開発したデータフロー基盤である Pratipad を用いる．クラウド層には，Elixir の Web アプリケーションフレームワークである Phoenix を用いて開発したサーバーアプリケーションを用いる．

Elixir は動的型付けの関数型プログラミング言語であり，Erlang/OTP の動作する仮想機械（Erlang VM）上で動作するよう設計されている．また，Erlang/OTP の提供する分散ネットワーク基盤も，Elixir から利用可能である．Nerves は，Elixir によって IoT デバイスを開発するためのプラットフォームであり，Elixir が Erlang VM 上で動作するのに必要十分かつ最小限の機能を持つ Linux によるファームウェアを提供し，Raspberry Pi [62] や BeagleBone [63] 等を用いた IoT デバイスの開発を迅速に行える仕組みを提供している．Elixir をプログラミング言語として選択することで，Nerves と Phoenix を用いて，Pratipad を媒介としてデバイス層，エッジ層，クラウド層のいずれをも同一の言語で開発することが可能となる．また，各層の間の通信プロトコルも，Erlang/OTP の提供する分散ネットワークプロトコルに統一できる．Erlang/OTP は大規模な分散システムの構築に利用されてきた実績があり，IoT システムを構成する言語・通信基盤として十分な堅牢性を有する [64].

各層の間の通信プロトコルは，セキュリティを担保できることが必須である．Erlang/OTP は，ノードと呼ばれる Erlang ランタイム間における通信によって分散ネットワークを実現する．提案手法では，デバイス層，エッジ層，クラウド層の各層においてノードを起動し，それぞれのノード間で通信することで IoT システムのデータフローを実現する．各層のノードが障害等により再起動した場合にはその間の通信が途絶し，メッセージの送信ロスが発生し得る．一方で，再起動後は各ノードが再接続をすることで，通信を再開できるよう実装している．ノード間通信においては，TLS（Transport Layer Security）を用いて通信内容を暗号

化することで，各層の間の通信経路における秘匿性を担保できる．また，クライアント証明書による認証を用いることで，不正なノードによる通信への介入を防ぐことができる．

3.2.2　多様かつ双方向性をもつデータ取得方式への対応

課題（2）は，選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるかということであった．そこで，本節ではいずれの方式にも対応しつつ双方向通信も可能にするプロトコルを，前述の Pratipad によって Erlang/OTP の分散ネットワーク基盤上に定めることで，課題を解決する手法を提案する．ノード間通信においては，相互に接続されたノード同士が Elixir のデータ構造である Atom で示される識別子によってメッセージの種別を指定することで，複数の種別のメッセージをやりとりできる．提案手法では，デバイスからのデータ取得においてどのデータ取得方式を用いるか（以下，これをモードと呼ぶ）を設定できるようにする．そして，モードごとのメッセージの種別を定義することによって，同じ基盤を用いながらいずれのデータ取得方式をも扱えるようにする．

push モードのシーケンス図を**図 3.1** に示す．各層のプログラムはそれぞれ独立に動作している．push 方式において，データフローに送り込まれるメッセージ数を決定するのはデバイス層である．まずは順方向（デバイス層からクラウド層への方向）のデータフローから説明する．デバイス層からメッセージを`:push_message`[3] によって送信することで，データフローを開始する．エッジ層は，受け取ったメッセージに対して変換・情報の加除・集約等を施した上で，`:forward_message` によりクラウド層へ送信する．次に逆方向（クラウド層からデバイス層への方向）のデータフローについて説明する．クラウド層からデバイス層へのアクチュエーション指示などのメッセージを送るためには，まずクラウド層が`:push_message` に引数を渡した上でエッジ層へメッセージを送る．エッジ層は，受け取ったメッセージを`:forward_message` によりエッジ層へ転送する．本章の想定する IoT システムは階層的なアーキテクチャーであり，順方向と逆方向のデータフローは非対称的である．そのため，本手法では逆方向のデータフローについては，エッジ層

[3] この識別子は，前述した Elixir の Atom によって表現されている．以下，同様である．

における処理を行わない．

pull モードのシーケンス図を**図 3.2** に示す．各層のプログラムがそれぞれ独立に動作しているのは push 方式同様である．pull 方式において，データフローに送り込まれるメッセージ数を決定するのはクラウド層である．順方向のデータフローでは，クラウド層からデバイス層へのメッセージ送信を指示するメッセージを:push_message に引数を渡した上で送信することで，データフローを開始する．エッジ層は，受け取ったメッセージを:forward_messge によりデバイス層へ転送する．デバイス層は，メッセージを:push_message により送信する．エッジ層は，受け取ったメッセージに対して変換・情報の加除・集約等を施した上で，:forward_message によりクラウド層へ送信する．逆方向のデータフローについては，push 方式と同様である．

demand モードのシーケンス図を**図 3.3** に示す．各層のプログラムがそれぞれ独立に動作しているのは前述 2 つの方式同様である．demand 方式において，データフローに送り込まれるメッセージ数を決定するのはエッジ層である．順方向のデータフローにおいて．エッジ層は現在処理されているメッセージ数があらかじめ定められた限度内に収まっているかどうかを確認する．もし限度内に収まっていたら処理の余力があると判断し，処理可能なメッセージ数の余力を引き下げた上で，デバイス層へ:pull_message によってメッセージの送信指示を行うことでデータフローを開始する．デバイス層は，:send_message によってエッジ層へメッセージを送信する．エッジ層は，受け取ったメッセージに対して変換・情報の加除・集約等を施した上で，:forward_message によりクラウド層へ送信する．処理が終わったら，エッジ層は処理可能なメッセージ数の余力を引き上げる．逆方向のデータフローについては，前述 2 つの方式同様である．

各モードのいずれにおいても，デバイス層のノードからエッジ層のノードへ，エッジ層のノードからクラウド層のノードへと，各層のノードが起動時に Node.connect/1 関数 [65] を用いて TCP 接続を確立することで，Erlang/OTP の分散ネットワークを形成する．各層のノード上で起動した Elixir プロセスは，自身のプロセスを示す一意な名前を自身のプロセス ID と紐付けて:global.register_name/2 [66] を用いて登録する．その名前を指定することで，各層間での通信が可能となる．本提案では，上述のようにデバイス層，エッジ層，クラウド層の方向でネットワー

クの経路が存在するが，その逆方向については必ずしも経路が存在しないネットワーク構成を前提とする．そうした構成はクラウド層においてデータを集約することを目的とする IoT システムにおいては一般的であり，提案に対して強い制約をもたらすものではないと考える．なお，逆方向への経路が存在しない場合でも，順方向で層間の接続が確立された後は，確立されたセッションを通じて逆方向へのデータ送信が可能となる[4]．

各層間の通信の実装を支援するために，Pratipad のクライアントライブラリとして筆者らが用意した Pratipad.Client [68] を用いることができる．各層においては，上述の過程を経て形成されたネットワーク上で，push，pull，demand のそれぞれのモードで通信を行うが，Pratipad.Client を用いることで，容易に各モードに対応する実装を行える．また，各層間の通信における接続先の名前の指定についても，Pratipad.Client 利用時の設定を通じて行う．さらに，Pratipad.Client は，Pratipad の動作するエッジ層と通信するデバイス層とクラウド層において，各モードのシーケンス図に示したそれぞれのモードにおけるメッセージ名に対応する関数を記述すると，その関数が返却したデータを用いて各層の通信を実現する機構を備えている．

3.2.3 可読性の高いデータフロー表現記法

課題（3）は，課題（2）に加えて，データフローの記述にはエッジ層におけるデータ処理の記述も必要であることから，IoT システムの階層的なアーキテクチャにおけるデータフローを見通し良く扱えるかということであった．そこで，本節では 3 層からなる IoT システムのデータフローを宣言的に記述した上で，処理と分離して記述できる記法を提案する．そのことで，データフローの全体を一望のもとに把握することが可能となる．記法の一覧は**表 3.1** の通りである．

提案する記法は，前述の Pratipad によって実装されており，Elixir のコードとして記述して実行できる．Elixir および Erlang/OTP においては，それらで書かれたコードは beam ファイルと呼称されるバイトコードにコンパイルされ，Erlang VM 上で実行される．そのため，Elixir のコードとして記述したデータフローにつ

[4] この件の技術的詳細については，筆者による技術記事 [67] を参照されたい．

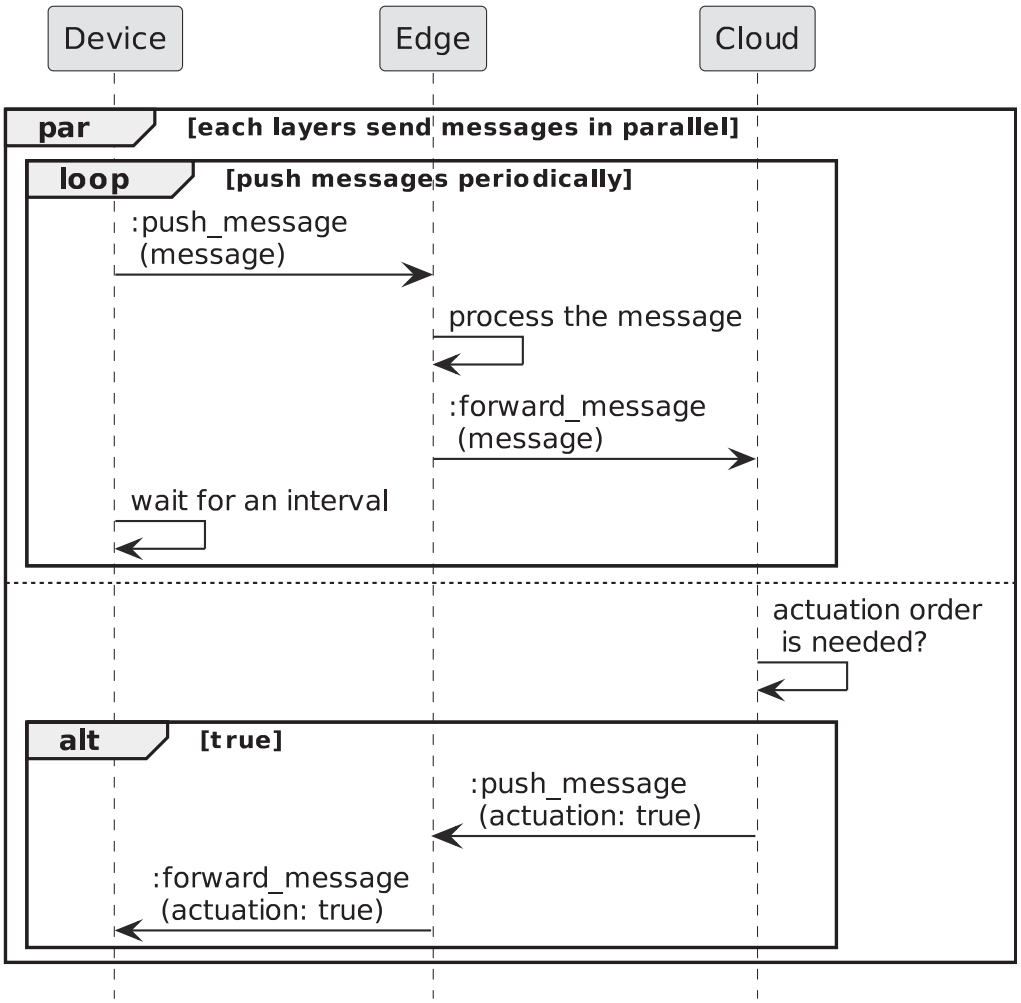

図 3.1: push モードのシーケンス図

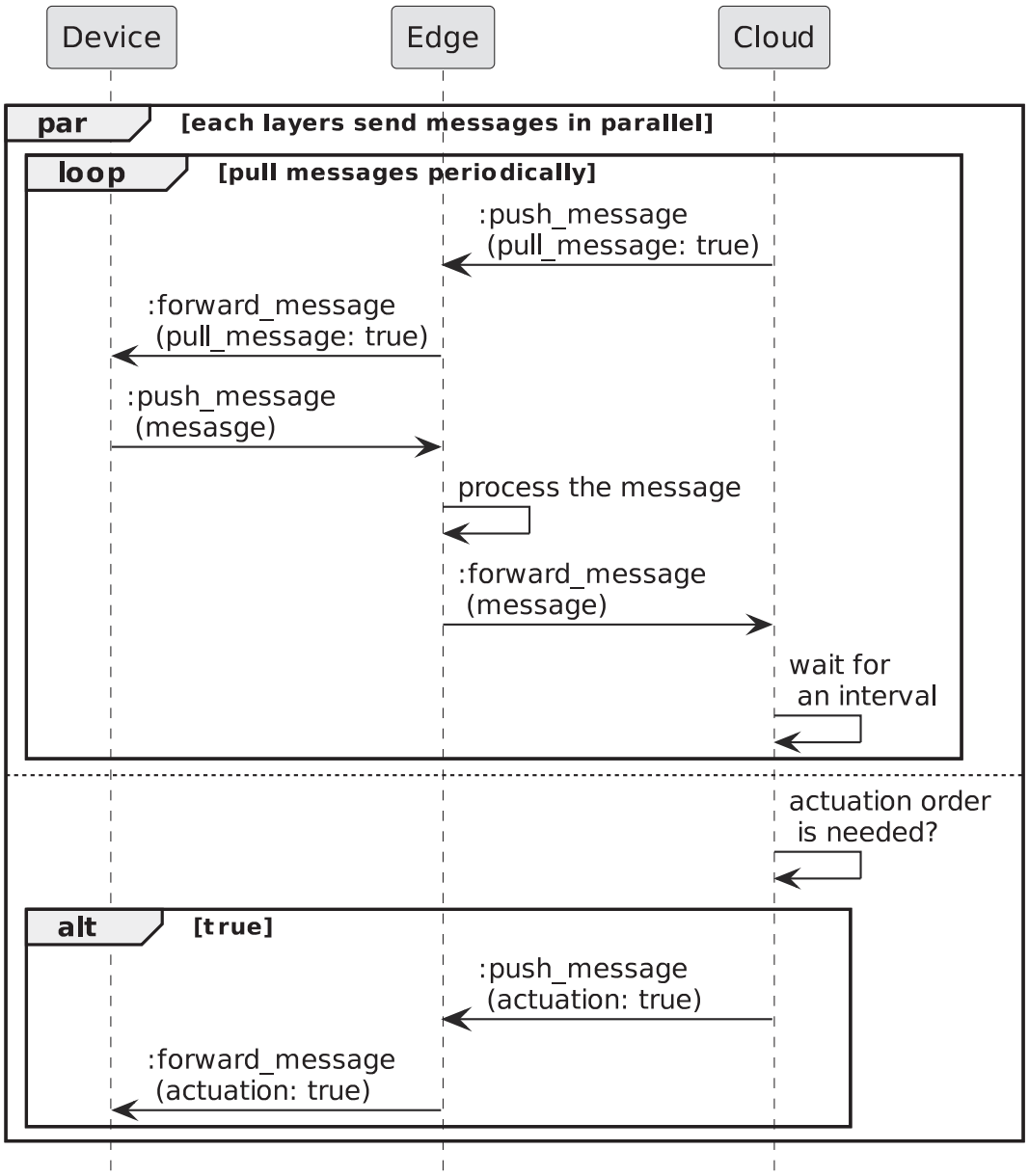

図 3.2: pull モードのシーケンス図

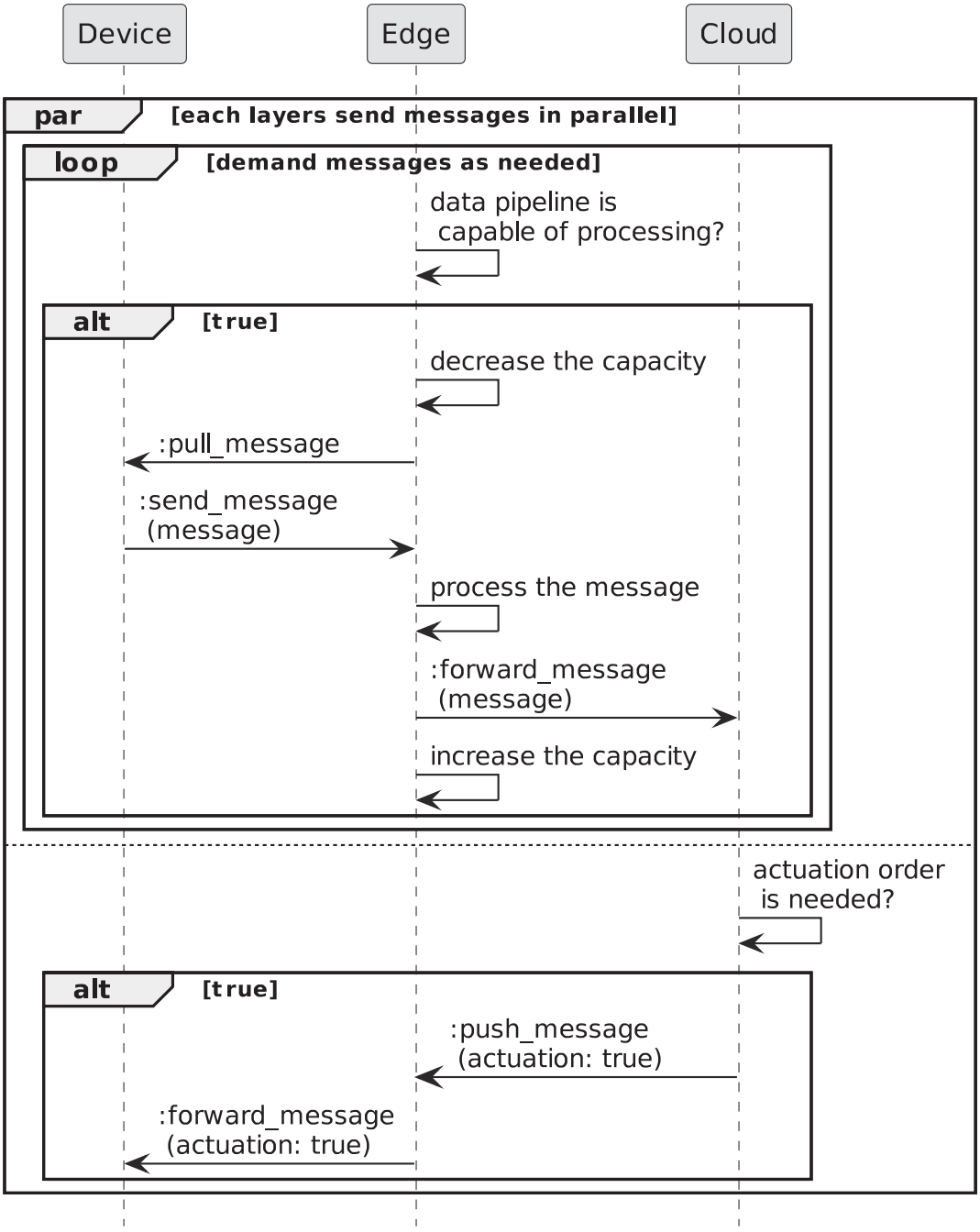

図 3.3: demand モードのシーケンス図

いても，Pratipad にロードされる形で他の依存ライブラリ等とともに同様にコンパイルが行われ，エッジ層で起動している Erlang VM において実行される[5]．

提案する記法には，入力，処理，方向，出力の 4 つの種別がある．入力記法は，前述の提案（2）で述べた 3 つのデータ取得方式を記述する記法であり，それぞれ push モード，pull モード，demand モードに対応する．処理記法は，エッジ層におけるデータ処理を，処理を担当するモジュール（以下，プロセッサーと呼ぶ）の名前を記述することで表現する記法である．単一の処理のみの場合は，対応するプロセッサー名をひとつだけ記述する．順次適用される処理内容の場合は，プロセッサー名を Elixir のリストによって列挙する．並列適用される処理内容の場合は，プロセッサー名を Elixir のタプルによって列挙する．順次適用は，メッセージ内容への変換・情報の加除等が以前の処理に依存する一連の処理に用いられ，処理後のメッセージが出力される．並列適用は，メッセージ内容を外部へ送信するといった，メッセージ内容への副作用のない独立して並列的に実行できる一連の処理に用いられ，入力されたメッセージがそのまま出力される．方向記法は，データフローが順方向のみの単方向であるか，逆方向にも対応した双方向であるかを記述する記法である．出力記法は，上述の記法に基づくデータフローの記述において，その終端を示すための記法である．

図 3.4 に，提案手法を用いたデータフローの記述例を示す．データフローは，任意の名前を持つモジュール内において，`Pratipad.Dataflow` の要求する `declare` 関数内に記述する必要がある．データフローを記述するためのモジュールでは，`use` マクロによって `Pratipad.Dataflow` を実装することを宣言し，`alias` マクロによって**表 3.1** に示した入力モードを示すモジュール名（`Push`, `Pull`, `Demand`）のうちから必要なものと，データフローの出力を示すモジュール名 Output を読み込む必要がある．

この例では，データフローへの入力は `Push` で示される push モードであり，前節の**図 3.1** で示したエッジ層への入力（図内の`:push_message`）にマッピングされている．また，このデータフロー記述は`<~>`で示される双方向のフローを表現している．方向記法は，Elixir のマクロによって DSL（Domain Specific Language）として実装されており，演算子同様に用いることができる．双方向のフローを記

[5] 3 層におけるコードはそれぞれ独立しているため，デバイス層とクラウド層を実装するコードについてもそれぞれビルドとデプロイが必要である．

述した場合は，クラウド層からエッジ層を通じて，デバイス層へ逆方向のメッセージ送信が発生し得る．そのため，デバイス層では前節の図 fig:sequence-push で示した逆方向のメッセージを受け取るための :forward_message 関数を実装する必要がある．なお，逆方向のデータフローについては，デバイスへのアクチュエーション指示を想定しており，エッジ層でのデータ処理は行わない．そのため，クラウド層から送信されたメッセージは，エッジ層からそのままデバイス層へ中継される．入力されたデータに対して，P1 で示されるプロセッサーの処理が適用される．後続の P2 および P3 についても同様に，プロセッサーによって順次処理が適用される．これらのプロセッサーは Elixir のプロセスとして起動し，エッジ層において実行される．

データフロー記述の最後にある Output は，エッジ層で処理されたメッセージがクラウド層へ送信されることを示す．データフローは，**図 3.4** のようにひとつのファイル内に記述される一方で，処理記法で記述されたプロセッサー名に関連する実装は，それぞれ個別のファイル内に記述される．**図 3.5** に，提案手法を用いたデータ処理の記述を示す．この例では，**図 3.4** のデータフロー内に記述されている P1 モジュールの処理の実装例を記述している．プロセッサーは，Pratipad.Processor ビヘイビア[6]の要求する process 関数を実装する必要がある．メッセージの処理時にこの関数が呼ばれることで，処理が行われる．このように，データフローの宣言と処理の詳細の記述を分離することで，複雑なデータフローを一望のもとに把握できる記法を実現できている．

ところで，デバイスが複数のセンサーを持ち，かつ，それぞれのセンサーについて異なるデータフローを構成することも考えられる．現状の記法の設計，および，Pratipad の実装においては，データフローはデバイスにつきひとつのみであることを前提としている．今後，複数のデータフローを扱えるよう拡張することが望ましい．一方で，単一メッセージ内に複数のセンサーデータをまとめて送信できるため，データフローへの入力方式が同一であれば，複数のセンサーデータを扱うことも可能である．

[6]ビヘイビア（Behaviour）とは，モジュールの中で実装しなければならない関数を定める機能である．Java のインターフェイスに相当する．

表 3.1: IoT システムのデータフローを記述するために Pratipad が提供する記法

種別	記法	概要
入力	`Push`	push モード
	`Pull`	pull モード
	`Demand`	demand モード
処理	`P`	単一のプロセッサー
	`[P1, P2, ... PN]`	順次適用する複数のプロセッサー
	`{P1, P2, ... PN}`	並列適用する複数のプロセッサー
方向	`~>`	データフローは単方向
	`<~>`	データフローは双方向
出力	`Output`	データフローの終端

```
1 defmodule Example.Dataflow do
2   use Pratipad.Dataflow
3   alias Pratipad.Dataflow.{Push, Output}
4
5   def declare() do
6     Push <~> P1 <~> P2 <~> P3 <~> Output
7   end
8 end
```

図 3.4: 提案手法を用いたデータフローの記述例

3.2.4 提案手法を実装する上での技術的課題

提案手法の要となる筆者が開発した Pratipad の実装においては，解決するべき技術的に困難な課題があった．まず，IoT システム特有の多様なデータ取得方式（push 方式，pull 方式，demand 方式）に対応し，かつ，階層間での双方向通信を Erlang 分散ネットワーク上で実現することが課題であった．そのため，既存のライブラリ Broadway [26] を拡張して上述の方式および双方向通信を実現するために，off_broadway_distribution [69] を開発し，提案手法の実装に組み込んだ．この実装には，上述のデータ取得方式への理解と実装技術はもとより，Erlang/OTP の分散ネットワークプロトコルに対する詳細な知識が必要であった．また，データフローの宣言的記述を可能にする DSL の設計・実装も技術的挑戦であった．処理の順次適用や並列適用，双方向通信などの複雑なデータフローを，開発者が直感的かつ簡潔に記述できるようにするために，Elixir のマクロ機能やビヘイビアを駆使して高度な抽象化を実現した．この実装には，メタプログラミング技法や Elixir

```
defmodule P1 do
  alias Pratipad.Processor
  use Processor

  @impl GenServer
  def init(initial_state) do
    %{:ok, initial_state}
  end

  @impl Processor
  def process(message, state) do
    # do something with the message
  end
end
```

図 3.5: 提案手法を用いたデータ処理の記述例

の機能や文法に対する深い知識が要求された．これらの技術的困難を克服することで、Pratipad は IoT システム全体を統合的に構築できる基盤となったと考える．

3.3 評価

本章では，提案手法の有効性と適用可能性を評価する．

3.3.1 有効性の評価

本節では，提案手法のそれぞれについて有効性の評価を行う．本節における評価の目的は，3.3 節で提案した手法が本章で扱う課題を解決すると同時に，実用的なシステムとして有用なものであることを検討することである．

提案（1）の評価

3.2.1 項では，各層の実装に用いるプログラミング言語として Elixir を選択した．本節では，選択したプログラミング言語とデータフロー基盤の Pratipad を用いて，実際に階層的なアーキテクチャからなる実用的な機能を持つ IoT システムを構築できることを示す．

提案手法を用いたIoTシステムの構築例を**図 3.6**に示す．このシステムは，住居の室内環境を計測し，作業の遂行に不適切な状況である場合に，利用者に対して窓を開けるよう促すためのものである．構築したシステムの前提として，デバイス層とエッジ層を構成するハードウェアは住居内のLANに接続され，互いに疎通可能であることを想定している．また，クラウド層への接続については，LAN内からWANを通じて接続が可能であり，WAN側からはLAN内のデバイス層・エッジ層への経路が存在しないことを想定している．

本システムは，デバイス層，エッジ層，クラウド層の3層からなるため，処理の流れをそれぞれの層に沿って説明する．

1. **デバイス層** 住居内の複数の部屋において，センサーを用いて室内環境（二酸化炭素濃度，温度，湿度，気圧）を計測し，エッジ層へ送信する．また，クラウド層からのアクチュエーション指示によって，利用者に室内環境の悪化と窓を開けることの必要を知らせるために，LEDを点滅させる．
2. **エッジ層** デバイス層から送信されたセンサーデータに対して，気象庁のホームページで配布されているオープンデータ[7]から取得した雨量データを付与した上で，クラウド層へ送信する．室内環境は部屋ごとに異なるが，雨量は住居のある地域の情報を得られれば実用上十分なため，エッジ層で付与することが効率的である．
3. **クラウド層** エッジ層から送信されたセンサーデータに基づき，二酸化炭素濃度が閾値を超える場合で，かつ，強い雨が降っていない場合に，利用者に対して窓を開けるよう促すためにデバイス層に対してアクチュエーション指示を行う．また，直近数時間のセンサーデータの推移をグラフ表示することで，利用者が室内環境の変化を確認できる機能も提供する．

本システムは，3層をそれぞれ異なるハードウェア，OSによって構成している（**表 3.2**）．一方で，各層を構成するソフトウェアはElixirを用いて書かれており，各層間の通信はErlang/OTPによる分散ネットワーク基盤を用いて行われている．また，通信経路はTLSで暗号化されており，クライアント証明書による認証を用いたセキュアな接続を実現している．本システムの実装においては，各層を構成

[7]気象庁｜最新の気象データ https://www.data.jma.go.jp/obd/stats/data/mdrr/

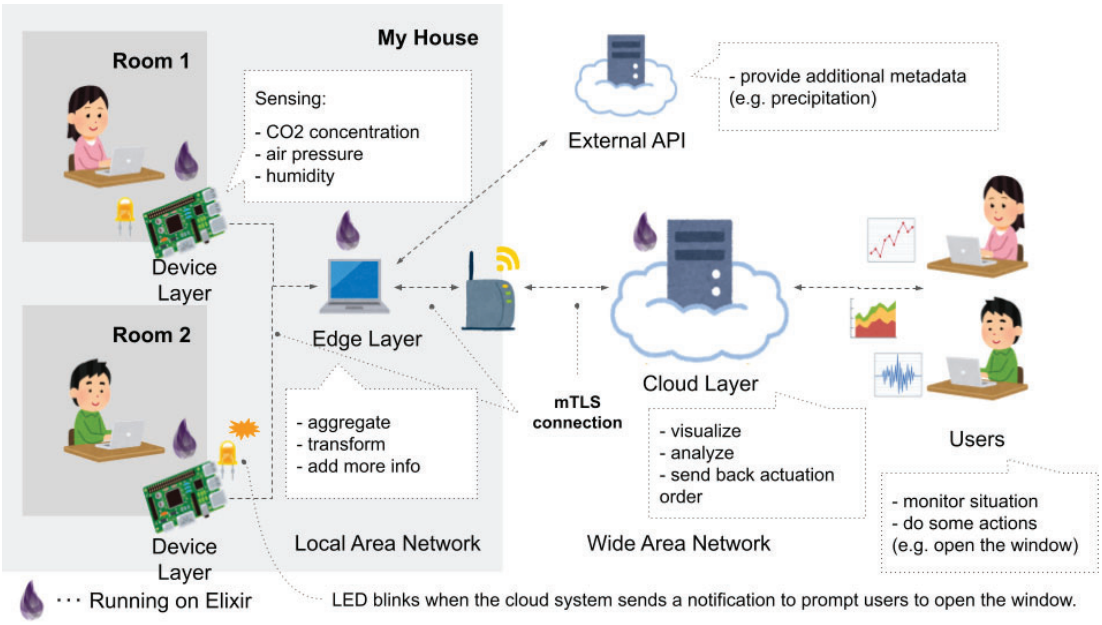

図 3.6: 提案手法を用いた IoT システムの構築例

表 3.2: IoT システムの構築例に用いた構成

階層	ハードウェア	OS
デバイス層	Raspberry Pi Zero W	Nerves（Linux ベース）
エッジ層	Raspberry Pi 4 Model B	Raspberry Pi OS
クラウド層	Google Cloud e2-small	Debian GNU/Linux

するソフトウェアの成果物に証明書を同梱し，Erlang/OTP が提供する TLS による通信機能 [70] を利用し，証明書を起動時に読み込む設定を施した．

上述の IoT システムの構築例により，提案手法は課題（1）を解決することを示した．加えて，Pratipad によるエッジ層におけるデータ処理を可能とすることで効率的なデータフローを実現した．さらに，各層の特性に応じて異なるハードウェア，OS を利用する 3 層を通じて提案手法を用いた IoT システムを実現できることも示した．提案手法は，単に IoT システム全体を構築可能であるという以上の利点をもたらしているといえる．このことから，提案手法は階層的なアーキテクチャからなる実用的な機能を持つ IoT システムを，単一のプログラミング言語によって構築可能な手法として有効であると評価できる．

提案（2）の評価

3.2 節では，Elixir を用いて多様かつ双方向性をもつデータ取得方式に対応できる基盤として，Pratipad を提案した．本節では，3.3.1 項で取り上げた IoT システムの構築例をもとに，データ取得方式とデータフローの双方向性について検討することで，提案手法の有効性を評価する．

IoT システムに求められる要件次第で，どのデータ取得方式に利点があるかは異なる．そのため，同じ 3 層からなる IoT システムであっても，目的によってデータ取得方式を使い分ける必要がある．

1. **push 方式** デバイスが取得可能なセンサーデータを余すことなく送信できる．IoT システムに対して，デバイス層において取得したデータの時間分解能の高さが求められる際に有効である．
2. **pull 方式** クラウド層が提供するユーザー向けのアプリケーションに必要なだけのデータを取得できる．IoT システムに対して，ユーザーへ提供するサービスレベルに応じてデータの流量を制御できることが求められる際に有効である．
3. **demand 方式** データ処理を行うエッジ層の余力に応じてデータフローの流量を制御しつつデータを取得できる．IoT システムに対して，リソース制約のもとでの高い可用性が求められる際に有効である．

データフローの双方向性についても，IoT システムの要件によって使い分ける必要がある．単方向のみに対応している IoT システムを，運用開始後に双方向に対応させることには困難が伴う．そのため，要件の変更を見据えて，容易に双方向性に対応できる方式を用いることが望ましい．

上述の整理と**図 3.6** で示した IoT システムの構築例とにより，提案手法は課題（2）を解決することを示した．提案手法は 3 つのデータ取得方式，および，データフローの単方向・双方向のいずれにも対応できる．加えて，それらを使い分けるために必要な作業は，データフロー記述の変更と，3.2 節で説明したデバイス層とクラウド層で Pratipad.Client を利用してデータ送信を実装する関数の変更のみである．そのことで，提案手法は多様かつ双方向性をもつデータ取得方式に対応できているという以上に，データフロー実装上の利点をもたらしている．このこ

とから，提案手法は様々な要件が求められる IoT システムの構築にとって有効な手法であると評価できる．

提案（3）の評価

3.3 節では，階層的なアーキテクチャにおけるデータフローを一望のもとに把握できる記法を提案した．本節では，提案手法を用いて 3.2 節で示したデータ取得方式および双方向通信，そして 3.3 節で示したエッジ層における処理の順次適用と並列適用を表現できることを示す．

提案手法は，push，pull，demand の 3 つのデータ取得方式に加えて，それぞれのデータ取得方式における単方向，双方向いずれについても表現できる．これらのうちで pull 方式は，クラウド層からデバイス層へのデータ送信指示に基づいてデバイス層がデータを送信するため，双方向通信が可能なことが前提である．そのため，データ取得方式と通信の方向の組み合わせは 5 通りとなる．**表 3.3** に示す通り，提案手法はその 5 通りについて全て表現可能である．また，提案手法は，エッジ層における処理の順次適用と並列適用についても，それぞれ単独であるいは組み合わせて表現できる．**表 3.4** では，順次適用のみ，並列適用のみ，それぞれを組み合わせた処理の記述について，それぞれまとめている[8]．

以上により，提案手法は課題（3）を解決することを示した．提案手法を用いると，データ取得方式，通信の双方向性，エッジ層における処理の組み合わせを表現できる．また，データフローと処理を分離して表現できるため，見通しの良い記述が可能である．さらに，提案手法の提供する記法の表現能力は，本項で整理した通り，データフローを構成する上で必要十分なパターンに対応可能なものである．このことから，提案手法は，階層的なアーキテクチャにおけるデータフローを表現できる記法として有効な手法であると評価できる．

考察

前項までで，提案手法が本章の対象とする課題をすべて解決し得るものであるという評価を行った．一方で，IoT システムを構築する上で用いる単一のプログ

[8]単一のプロセッサーの例については，**表 3.3** に挙げているため，省略している．

表 3.3: 提案手法を用いた各データ取得方式に対応するデータフローの記述

方式	方向	記法
push	単方向	`Push ~> P ~> Output`
	双方向	`Push <~> P <~> Output`
pull	双方向	`Pull <~> P <~> Output`
demand	単方向	`Demand ~> P ~> Output`
	双方向	`Demand <~> P <~> Output`

表 3.4: 提案手法を用いたエッジ層におけるプロセッサーの記述

方式	記法
順次	`Push ~> [P1, P2] ~> Output`
並列	`Push ~> {P1, P2} ~> Output`
順次＋並列	`Push ~> [P1, P2] ~> {P3, P4} ~> Output`
並列＋順次	`Push ~> {P1, P2} ~> [P3, P4] ~> Output`

ラミング言語として，Elixir を選択することによる制約も存在する．具体的には，提案手法が前提とするハードウェアスペックに関する制約と，Elixir および分散 Erlang ネットワークプロトコルという特定のプログラミング言語および通信プロトコルを選択することによる制約とが挙げられる．また，3.3.1 目で評価したデータフローを記述する記法について，2.1.3 項で検討した関連研究との比較において，本章の提案に対して取り入れるべき点も存在する．

プログラミング言語として Elixir を選択したことによる制約に関する考察について述べる．3.1 節で述べた通り，Elixir は Erlang/OTP の動作する仮想機械（Erlang VM）上で動作する言語である．そのため，Elixir の実行には仮想機械方式での動作が可能となるスペックのハードウェアが必要となる．提案手法においてデバイス層の実装に用いた Nerves は Raspberry Pi や BeagleBone に対応している [71] が，それらはいずれも [37] がハイエンドな IoT デバイスとして分類しているものである．現在のところ Nerves は，ミドルエンドやローエンドのデバイスには対応していない．そのため，提案手法を用いた IoT システムの構築には，ハイエンドな IoT デバイスの使用が前提となる．一方で，そのような制約が今後緩和されていく見込みもある．[72] は AtomVM という IoT デバイス上で動作することに特化した Erlang VM を開発しており，[37] がミドルエンドと分類する ESP8266 の後継であ

る ESP32 上で動作する．よって，今後，提案方式がより低いスペックのハードウェア上でも適用できる可能性があり，提案手法が前提とするハードウェアスペックに関する制約が緩和されることが期待できる．

また，分散 Erlang ネットワークプロトコル [73] は公開された仕様であり，任意の言語で実装可能である．実際に，プログラミング言語 Go で分散 Erlang ネットワークプロトコルを実装した Ergo [74] というプロジェクトが存在しており，本提案手法同様に Go のみを用いて階層的な IoT システムを構築できる可能性がある．このように，提案手法は Elixir という特定の言語に閉じない一般的な手法としての発展へと開かれているといえる．

関連研究との比較において，本章で述べた提案手法に対して取り入れるべき点について述べる．2.1.3 項で関連研究として取り上げた Node-RED と DDF について，2.1.4 項ではそれらは本章が対象とする 3 つの課題すべてを解決できるものではないとした．一方で，それらの手法によって GUI 上でノードを繋げていくことでデータフローを表現できることは，データフローを見通しよく記述することに大きく寄与し得る．提案手法の現在の実装では，記法を Elixir のコードとして直接書くことでデータフローを記述する方法を取っているが，関連研究の利点を取り入れることで改善が可能であると考える．すなわち，GUI によるデータフロー表現を本章の提案する記法を用いたコードとしても表現できるようにし，それらを同一のデータフローを表現するものとして自由に切り替えられるようにすることで，両者の利点を両立し本章で述べた提案手法の価値を高め得る．

3.3.2 適用可能性の評価

実験方法

3.3.1 項では，提案手法が本章の目的とする IoT システムを単一のプログラミング言語で構築できるデータフロー基盤を実現し得ることを，有効性を評価することで示した．一方で，提案手法を適用して実用的な規模の IoT システムを構築するためには，提案手法がその実現に耐え得る性能を発揮できるか否かについても示す必要がある．そこで，提案手法の適用可能性を評価するために，性能に関する実験を行う．

表 3.5: 実験環境

階層	ハードウェア	OS
デバイス層	iMac (24-inch, M1, 2021)	macOS
エッジ層	Raspberry Pi 4 Model B	Raspberry Pi OS
クラウド層	Google Cloud e2-small	Debian GNU/Linux

実験環境を**表 3.5** に示す．実験では，3.2.2 項で述べたデータ取得方式について，秒間スループットの推移を計測する．また，その間のリソース使用量について確認するため，CPU 使用率とメモリ使用率を取得する．実験の対象とするのは，push 方式と demand 方式とする．push 方式は，デバイス層からできるだけ多くのデータを取得するための方式であるため，できるだけスループットが高いことが望ましい．一方で，demand 方式は，リソース使用量を抑えつつエッジ層の余力の範囲でできるだけスループットが高いことが望ましい．そのため，実験ではスループットとリソース使用量を確認する．pull 方式は，クラウド層での需要に応じてメッセージ送信を要求する方式であり，スループットの最大化を目的とする方式ではないため，実験の対象としない．

実験の対象とするそれぞれの方式で動作するデータフロー基盤に対して，メッセージを送信するデバイス群をシミュレートするメッセージ送信用の Elixir プロセスを評価環境のデバイス層に見立てたホスト上で起動する．そして，1 デバイスに相当する 1 プロセスあたり毎秒 1 回程度メッセージが送信される頻度で，10 分間データフローにメッセージを流し続ける．すなわち，1,000 プロセスから送信されるメッセージ数は，毎秒 1,000 となる．push 方式では，それぞれのプロセスが 1 秒に 1 回メッセージを送信する．demand 方式では，1 秒間にデータフローで処理するメッセージ数の上限をプロセス数と同値とすると同時に，データフロー内の未処理メッセージ数がプロセス数の半分を下回った際に余力があるとして，メッセージ送信要求を行う設定とする．

```
1 Push ~> Precipitation ~> Output
```

図 3.7: 実験に用いる push 方式のデータフロー定義

実験環境のネットワーク環境およびデータフロー基盤における処理内容は，3.3.1

```
1 Demand ~> Precipitation ~> Output
```

図 3.8: 実験に用いる demand 方式のデータフロー定義

項で示した IoT システムの構築例と同一である．雨量データは外部 API へのリクエストにより取得し 30 分間キャッシュするため，実験中は外部 API へのリクエストが行われるのは一度のみである．また，本実験ではデバイス層からのデータ取得についてのスループットを計測するため，データフローの通信方向を片方向とする．以上を踏まえた push 方式と demand 方式のそれぞれのデータフロー定義は**図 3.7** および**図 3.8** の通りである．図内の `Precipitation` が，上述の雨量データを付与する処理を行うモジュール名を表している．

なお，提案手法の性能限界をあらかじめ把握するために，予備実験を行った．push 方式では，接続するプロセス数が 8,000 から 9,000 程度で実験環境に構築したシステムの動作が不安定となったため，継続的に安定して動作する 5,000 プロセスまでの実験を行う．demand 方式では，接続するプロセス数が 5,000 程度での動作が不安定となったため，継続的に安定して動作する 1,000 プロセスまでの実験を行う．両者で安定動作するプロセス数の上限に差があるのは，demand 方式は 1 メッセージの取得あたりに，push 方式と違って，2 回の通信が発生するためであると考えられる．

実験結果

3.3.2 目で述べた実験方法について，異なるプロセス数で行なった実験結果を**表 3.6** に示す．

表 3.6: 実験結果

方式	プロセス数	総メッセージ数	スループット（秒）	平均 CPU 使用率（%）	平均メモリ使用率（%）
push	10	6,000	10	1.11	3.48
demand		5,952	9.92	1.10	3.59
push	100	60,000	100	2.38	3.61
demand		59,720	99.53	1.83	3.74
push	1,000	600,000	1,000	7.08	3.83
demand		594,167	990.27	6.88	4.45
push	5,000	3,000,000	5,000	26.20	3.96
demand		-	-	-	-

いずれの実験においても，総メッセージ数は実験で設定したプロセス数に 10×60（10 分間）を乗じた値の近傍となっており，また，スループットは実験で設定したプロセス数の増加に応じた結果となった．すなわち，push 方式においては 5,000 プロセスまで，demand 方式においては 1,000 プロセスまでの，1 プロセスあたり 1 秒に 1 回程度のメッセージ送信を処理可能であることが示された．また，いずれの実験においても，平均 CPU 使用率および平均メモリ使用率はリソース総量に比べて低い値に止まっており，リソース使用率が逼迫している様子も見られない．予備実験で検討した性能限界は，接続するプロセス数の限界に依存するものと考えられる．

ここで，push 方式と demand 方式とで総メッセージ数とスループットに差異があるのは，push 方式と demand 方式のメッセージ送信における差異による．すなわち，push 方式ではデバイス層を起点としてメッセージを送信するためプロセスごとにメッセージ送信数を制御できるが，demand 方式ではエッジ層からの送信要求に応じて各プロセスがメッセージを送信するため，評価環境でそれぞれのプロセスが起動する時間のずれによって，後から起動したプロセスが送信する総メッセージ数が少なくなってしまうからである．これは実験上の制約に基づく差異であり，実際の IoT システムにおいて個別のデバイス上で常にプロセスが動作している環境ではそのような差異は発生しない．

それぞれの実験を実行している 10 分間における CPU 使用率およびメモリ使用率を `sar` コマンドで 5 秒ごとに取得し，プロセス数ごとに push 方式と demand 方式ごとに比較したグラフが**図 3.9** から**図 3.14** である．CPU 使用率およびメモリ使用率のいずれにおいてもリソース使用がスパイクしている様子は見られず，安定的にデータフローが動作していることが見て取れる．

図 3.10 に見られる通り，100 プロセスの場合に demand 方式の CPU 使用率が push 方式に比べて低い結果となった．3.3.1 目で述べた通り，demand 方式にはデータ処理を行うエッジ層の余力に応じてデータフローの流量を制御しつつデータを取得できる利点がある．実験環境で用いたハードウェアスペックにおいては，100 プロセス程度の接続による秒間スループット 100 程度までの流量において，その利点が活かせることが見て取れる．秒間スループットが 10 の場合は流量が少ないため差が見られず，1,000 の場合は余力に乏しく流量制御が十分に行えないため，push

図 3.9: CPU 使用率の比較（10 プロセス）

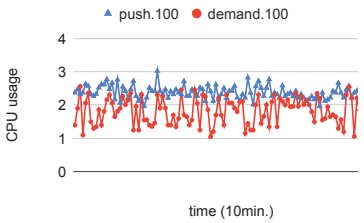

図 3.10: CPU 使用率の比較（100 プロセス）

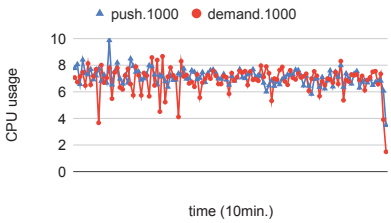

図 3.11: CPU 使用率の比較（1,000 プロセス）

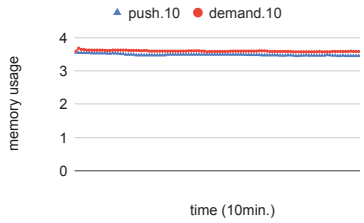

図 3.12: メモリ使用率の比較（10 プロセス）

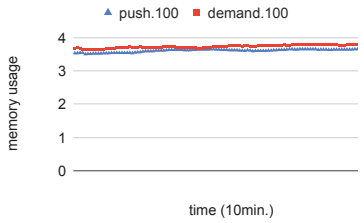

図 3.13: メモリ使用率の比較（100 プロセス）

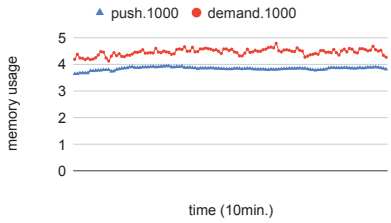

図 3.14: メモリ使用率の比較（1,000 プロセス）

方式同様の CPU 使用率となっている．一方で，**図 3.14** に見られる通り，1,000 プロセスの場合のメモリ使用率は，demand 方式が push 方式よりも高い結果となった．これは，demand 方式が一定程度メッセージをデータフロー内にバッファリングする方式であることによる．

考察

前項では，提案手法が秒間スループットにおいて，push 方式では 5,000 メッセージ，demand 方式では 1,000 メッセージを安定して処理できることを示した．3.3.1 目で示した IoT システムの構築例のような室内の二酸化炭素濃度を計測するシステムを例に取ると，提案手法では 1 デバイスを一部屋に配置するとして，push 方式では 5,000 部屋，demand 方式でも 1,000 部屋に設置できる規模の IoT システムを構築可能である．これは，4 部屋からなる住居が 1 フロアに 10 戸ある構成のマンションを想定すると，25〜125 階建程度の規模になる．また，室内環境を計測する IoT システムでは 1 秒に 1 メッセージよりも少ない頻度でのメッセージ送信で十分であろうことから，提案手法はより大規模な用途に用いることができる．よって，提案手法は実用的な規模の IoT システムの実現に足る性能を持つと評価できる．

また，提案手法には性能面で改良の余地がある．提案手法ではメッセージを分散 Erlang ネットワーク経由で送受信するために，筆者が開発したライブラリ [69] を用いている．現状の実装ではデータフローにおいてメッセージを受信するのは単一のプロセスのみである．よって，これを複数のプロセスによって処理する実装に改修すれば，実験で得られたより高いスループットを見込めるものと考える．

3.4 本章のまとめ

本章では，階層的なアーキテクチャからなる IoT システム全体を，単一のプログラミング言語で統合的に構築できるデータフロー基盤を提案した．その実現のために解決すべき課題として，(1) IoT システム全体を構築可能なプログラミング言語としてどの言語を選択するか，(2) 選択したプログラミング言語によって多様かつ双方向性をもつデータ取得方式に対応できるか，(3) IoT システムの階層的なアーキテクチャにおけるデータフローを見通し良く扱えるか，の 3 点を提示した．各課題に対して (1) 各層の実装に用いるプログラミング言語として Elixir を選択する，(2) Elixir を用いて多様かつ双方向性をもつデータ取得方式に対応できる基盤として Pratipad を提案する，(3) Pratipad において階層的なアーキテクチャにおけるデータフローを一望のもとに把握できる記法を提供する，という 3 点の提案手法により解決を図った．提案手法について，(1) 提案手法を用いて 3 層構造を持つ実用的な機能を持つ IoT システムを構築し，(2) 提案手法が 3 種類のデータ取得方式および双方向通信の全てを使い分けることができることを示し，(3) 提案手法の提供する記法がデータフローと処理を分離して記述でき，必要なデータフローを十分に表現できることを示すことで，提案手法の有効性を評価した．また，提案手法を用いた実験環境上で，メッセージ数を増加させながらスループットおよびリソース使用率を計測することで，提案手法が実用的な規模の IoT システムの構築に適用可能であることを評価した．

今後の展望としては，デバイスを実装するコードの変更内容を含む更新データや，デバイス層での機械学習に基づく推論のための学習済みモデルの更新データなどもまた，Pratipad のデータフローを通じてやり取りされるデータとして扱えるようにしたい．そのことで，階層的なアーキテクチャからなる IoT システムに

おけるデータフローを，本論文で扱ったデバイス層から取得したデータやクラウド層からの操作指示以外の内容を取り扱えるよう拡張でき，IoT システムの運用により資するものに発展できると考える．また，データフローを構成するデバイス数の増加により，システムの運用における負担も増加することが想定される．その場合に備えて，データフローに接続しているデバイスの情報を管理・参照できる機能を Pratipad に実装し，認証に用いるクライアント証明書の配布について効率的な方式を提案・実装することで，システム運用面における負担増加を低減し，本章で述べた提案手法の価値を高めることに寄与し得ると考える．

第4章　動的な可変性の導入

本章では，IoT デバイスへのコード変更の動的適用手法を提案・実装することで，IoT システムの開発に対して動的な可変性を導入する．

4.1　はじめに

IoT デバイスは年々増え続け，2030 年にはその数が 1250 億に達すると見込む調査報告がある [75]．また同調査報告は，スマートホーム機器のような家庭での利用はもとより，医療用途や製造業等における産業用途，エネルギーやモビリティに関わる社会インフラ基盤にいたるまで，多岐にわたる領域において IoT の活用が進むとしている．それらの増え続ける多様な IoT デバイスの需要を満たすためには，開発者が IoT デバイスを開発する効率を向上させることが必要である．

ハードウェアおよびソフトウェアの両面にわたる開発効率の向上のため，IoT デバイスの開発プラットフォームが多数現れている．ESP8688（およびその後継の ESP32）による開発キット [76]，Arduino [77]，Raspberry Pi [62]，Beagle Bone [78]，NVIDIA Jetson [79] 等がその例である．それらのプラットフォームは，ハードウェア開発効率向上のため，マイクロプロセッサや電源とともに，センシングやアクチュエーションのための様々な規格に対応する入出力インタフェイスや，Wi-Fi や Bluetooth モジュールといったネットワークインタフェイスを提供している．ソフトウェア開発効率向上のため，開発元やオープンソースソフトウェア（OSS）等により Arduino IDE [80]，ESP-IDF [81]，Platform.io [82] といった統合的な開発環境が提供されており，また，それらは開発者によるコードの変更を有線あるいは無線ネットワークを経由して行える OTA（over-the-air）による更新機能を備えている．上述した IoT デバイスを構成するハードウェアおよびソフトウェア，それら

に対応する開発プラットフォームを**表 4.1** の通り整理できる [83, p.180][1]，[32][2].

表 4.1 に示した IoT アプリケーションの開発には，パーソナルコンピュータ向けのアプリケーションや Web アプリケーション等の開発とは異なる困難さがある．パーソナルコンピュータ向けのアプリケーション開発においては，開発者の使用するホストとターゲットとなるホストのプラットフォームおよびアーキテクチャが同一である場合，開発者の使用するホスト上で動作確認を行える．また，Web アプリケーション開発においては，開発者の使用するホストとターゲットとなるホストのプラットフォームやアーキテクチャが異なったとしても，VirtualBox [84] や Docker [85] 等の仮想化技術を用いることで，開発者のホスト内で動作確認を完結する手法が発展している．一方で，IoT アプリケーションの開発においては，開発者が変更したコードの動作を確認するためには，なんらかの方法で変更内容をターゲットとなるデバイスへ適用し，IoT アプリケーションの動作を変更する必要がある．ターゲットとなるマシンのプラットフォームやアーキテクチャに加えて，操作対象となるハードウェア構成が開発者の使用するホストとは異なるためである．さらに，IoT アプリケーションのプロトタイピング時や開発時には，小規模な変更による頻繁な更新を必要とする．そのため，IoT アプリケーションの開発効率の向上を実現するには，開発者によるコードの変更を効率的にデバイスへ適用することが課題となる．

開発者によるコードの変更をデバイスへ適用する方式については，これまで多くの先行研究が行われている．[33, 34] は，コードの変更をデバイス上へ適用する方式を，書き換え対象に基づき（1）IoT アプリケーションの実装に用いられるスクリプト言語の特性を利用した動的なコード書き換え，(2）デバイス上で動作する仮想機械内で実行されるコードの書き換え，(3）デバイス上のファームウェアイメージの書き換え，(4）デバイス上のネイティブコードへの動的リンクの書き換えの 4 つに分類している．また，コードの変更をデバイスへ適用する状況について [33] は，デバイスの配備前における開発・検証，および，デバイスの配備後における機能追加・更新・拡張の 2 つに分類しており，それぞれについて更新頻度を整理している．それらの先行研究は，利用可能なネットワーク帯域，ハードウェアス

[1] ハードウェアを構成する要素として電源を [83] の定義に加えた．

[2] IoT の文脈では，IoT アプリケーションはエンドユーザ向けのアプリケーションを指す [8] が，本章では [32] に従い IoT デバイス内アプリケーションを IoT アプリケーションと呼称する．

表 4.1: IoT デバイスを構成するハードウェアおよびソフトウェア，それらに対応するプラットフォームの整理

	Components	Platforms
Hardware	Sensors Actuators Microprocessors Network Interfaces Power Supply	[62, 76–79]
Software	IoT Applications	[80–82]
	Network Protocol Stack	
	Operating System	—
	Hardware Drivers	

ペック，電源容量・確保において制約の厳しい IoT デバイスに対して，いかにして安全かつ効率のよい更新が可能であるかを主たる課題としている．そのため，上述の（1）および（2）についてはそれらの制約のもとでの実現は困難であるとして，検討対象外としている [33, 34]．また，更新の目的をデバイスの配備後におけるセキュリティ対策，バグ修正，機能拡張においているため，本章が課題とする IoT アプリケーション開発効率の向上は目的とされていない．

本章は，IoT アプリケーションの開発効率の向上という観点から，先行研究におけるコードの変更をデバイスへ適用する方式を，書き換え対象に基づき（1）ファームウェアイメージの全体を適用する方式,（2）ファームウェアイメージの差分を適用する方式,（3）アプリケーションコードを動的に適用する方式の 3 つに分類し直す．ファームウェアイメージの書き換えに基づく更新について，差分による効率的な更新を実現する研究が進んでいる [36]．そのため，全体を更新する方式を（1），差分を更新する方式を（2）として 2 つに分けて検討することが妥当である．IoT アプリケーションの開発効率を向上を目的として，前述の IoT デバイスの開発プラットフォームを用いたプロトタイピングが広く行われている．それらのプラットフォームは，開発者によるコードの変更をデバイスへ迅速に適用するために，有線あるいは無線ネットワークを経由したアップデートの方法を提供している．また，Raspberry Pi や Beagle Bone，NVIDIA Jetson などのように，Linux ベースのファームウェアが動作するハードウェアスペックを備えたプラットフォームも

存在している．IoT アプリケーションのプロトタイピング時や開発時においては，スクリプト言語や仮想機械上で動作する言語のような，ネイティブコードと比較してリソースを相対的に多く要求する言語を用いることは，開発効率の向上への寄与を目的として許容可能である．そのため，更新対象としての IoT アプリケーションを構成するコードは，ネイティブコードはもとより，スクリプト言語や仮想機械上で動作する言語による動的な性質を持つコードであっても，同等の機能を果たすとみなせる．よって，IoT アプリケーションの開発効率の向上という観点からは，アプリケーションのコードの変更をデバイスへ動的に適用する方式を，上述の方式（3）にまとめて検討することが妥当である．

本章は，IoT アプリケーションの開発効率の向上を目的とした，開発者によるコードの変更を IoT アプリケーションを構成するコードに対して動的に適用する方式について提案する．また，先行研究における，デバイスの配備後という強い制約下での更新方式という観点においては十分には検討されてこなかった，スクリプト言語や仮想機械上で動作する言語による動的な性質を持つコードを用いて提案方式を実装する．具体的には，開発者により変更された IoT アプリケーションを部分的に構成するコードを，ファームウェアイメージの生成を行うことなくネットワークを経由してデバイスに適用した上で IoT アプリケーションの更新を行う．提案方式の実装には，Elixir [52] を用いる．IoT デバイスの開発プラットフォームである Nerves [53] は，仮想機械上で動作する Elixir を開発言語として採用しているため，提案方式を仮想機械上で動作する言語を用いて実現できることに加えて，デバイス上のファームウェアイメージの全体および差分による更新に対応しているため，本章の整理した方式を網羅できるからである．検証に際しては，デバイス上のファームウェアイメージの書き換えを全体として行う方式と部分的に行う方式とをベースラインとして，更新に要する時間を提案方式の実装と比較検討した．その結果，既存方式がそれぞれ 66.88 秒，70.63 秒であったのに対して，提案方式では 3.30 秒で開発者によるコードの変更をデバイスへ適用することができ，95% の速度向上を実現できた．

4.2 提案手法と実装

4.2.1 提案手法

本章は，IoT アプリケーションの開発時における開発効率の向上を目的とした，開発者によるコードの変更を IoT アプリケーションを構成するコードに対して動的に適用する方式について提案する．先行研究においても，2.2 節で見た通り，アプリケーションコードを動的に適用する方式自体は提案されてきた．しかし，その目的はデバイスの配備後という制約下での更新方式という観点が主であり，開発効率の向上という観点は検討されてこなかった．また，利用可能なネットワーク帯域，ハードウェアスペック，電源容量・確保等において制約の厳しい IoT デバイスにおいては，必要とするリソースが相対的に大きくなるスクリプト言語や仮想機械上で動作する言語のような動的な性質を持つ言語は検討外とされてきた．本章では，IoT アプリケーションのプロトタイピング時や開発時という状況を前提とすることで，開発効率の向上のためにハードウェアを比較的自由に選択した上でリソース要求の大きな言語を用いることができる方式として，アプリケーションコードを動的に適用する方式を提案し，その有用性を主張する．

図 4.1 は，2.2.3 項で整理した（1）ファームウェアイメージの全体を適用する方式，および，(2）ファームウェアイメージの差分を適用する方式の処理の流れを示している．まず，開発者によるコードの変更に基づき，新たなファームウェアイメージを生成するフェーズが実行される．ファームウェアイメージの全体であるか差分であるかの違いはあるが，ビルドフェーズがあること自体は変わらない．次に，ファームウェアイメージをデバイスへ送信し適用するフェーズが実行される．ここでは，ネットワークを経由して送信することを前提としている．最後に，新しいファームウェアイメージをロードするためにデバイスが再起動するフェーズが実行される．

図 4.2 は，提案方式の処理の流れを示している．開発者によるコードの変更に基づき，IoT アプリケーションを実行するランタイムに対してコードを送信した上で，デバイス上で動作する IoT アプリケーションを動作させたまま変更されたコードを適用することで，アプリケーションの更新を行う．**図 4.1** とは異なり，コードを逐次的に送信・適用するため，ファームウェアイメージのようなひとまとま

りの成果物を生成・送信するフェーズは存在しない．また，IoT アプリケーションを実行するランタイムにおけるコードの適用であるため，IoT デバイス自体の再起動を要しない．

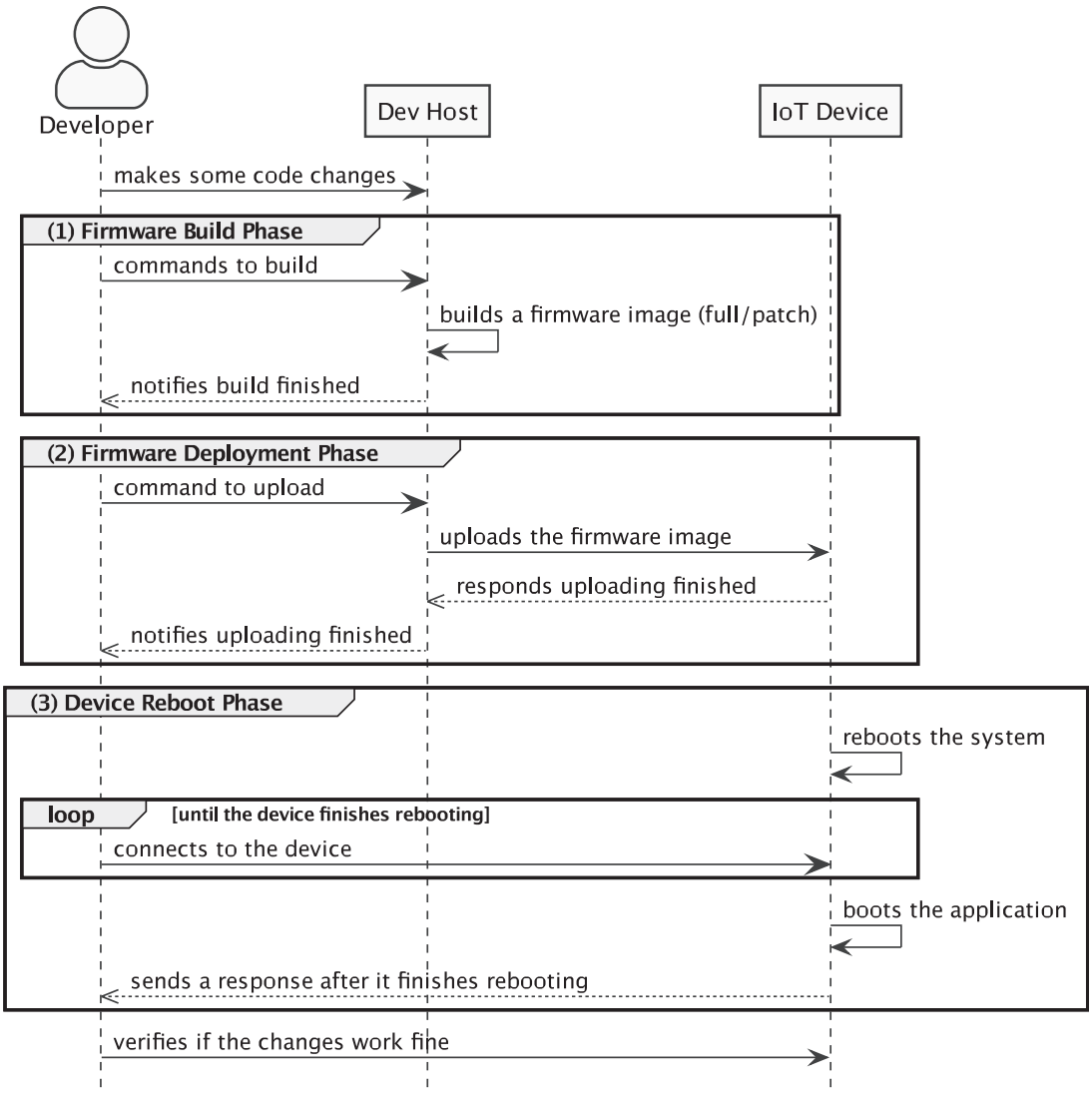

図 4.1: 既存方式のシーケンス図

4.2.2 実装

提案方式の実装には，プログラミング言語として Elixir [52] を，IoT デバイスの開発プラットフォームとして Nerves [53] を用いる．Elixir は動的型付けの関数型言語であり，大規模な並行プロセス動作を可能にするため，同様の目的で開発さ

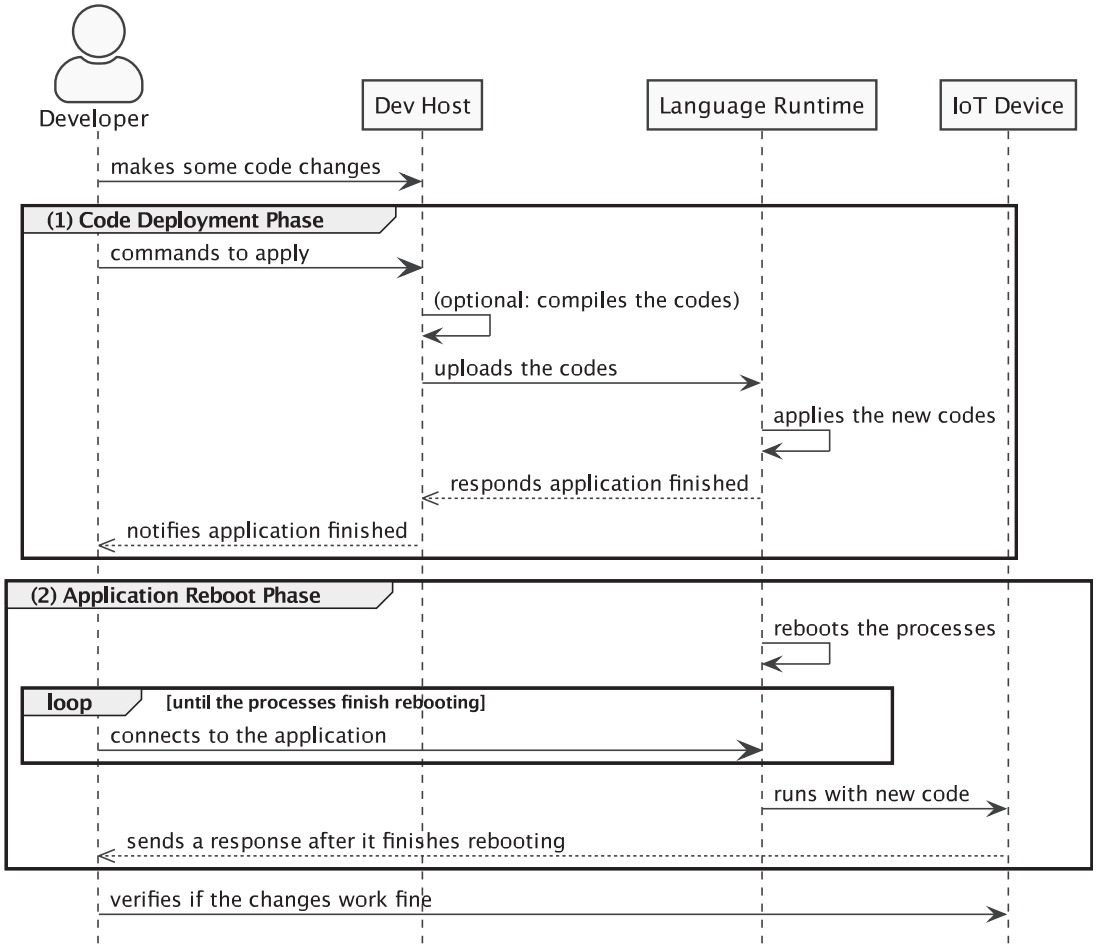

図 4.2: 提案方式のシーケンス図

れた言語システムである Erlang/OTP [57] の動作する仮想機械 Erlang VM 上で動作するよう設計されている．Nerves は，Elixir を Erlang VM 上で動作するのに必要十分なサイズの Linux によるファームウェアを提供するプラットフォームであり，Raspberry Pi や Beagle Bone 等を用いて IoT デバイスのプロトタイピングや開発を迅速に行える仕組みを提供している．また，仮想機械上で動作する Elixir を開発言語として採用しているため，アプリケーションコードを動的に適用する方式を実現できる．さらに，ファームウェアイメージの全体を適用する方式に加えてファームウェアイメージの差分を適用する方式にも対応しているため，本章の整理した方式を網羅できる．以上の理由から，本章では Elixir と Nerves を用いて開発した IoT アプリケーションを対象に，開発者によるコードの変更をデバイスに対して動的に適用する方式を実装した．

プログラミング言語 Elixir の実行基盤となる仮想機械である Erlang VM は，実行時にオブジェクトコードを動的に置き換えることができる機能を提供している [86]．コードの置き換えは，Erlang VM 上で動作するノードと呼ばれるプロセスを通じて行われる．ノードは遠隔手続き呼び出し（RPC: Remote Procedure Call）に対応しており，遠隔のホスト上で動作するノード上に対して RPC を実行することで，そのホスト上の Erlang VM への操作を実行できる．その機能を用いて，本実装では IoT デバイス上で動作するアプリケーションの動作を変更するためのコードの適用を実現する．具体的には，RPC を通じて Elixir から Erlang の `code:load_binary/3` 関数をモジュール名，ファイル名，オブジェクトコードのバイナリを引数として呼び出すことで，Erlang VM 上にロードされたコードを置き換える．

本実装では，IoT デバイス上で動作する Erlang VM 内で動作するノードへ開発者の用いるホストから接続し，上述の RPC を通じて Erlang VM 上で動作するコードの置き換えを行う．Erlang VM 上で動作するオブジェクトコードの置き換えが起こった際，古いコードには `old`，新しいコードには `current` というラベルが付され区別される．その状況で，再度コードの置き換えが起こると，`old` のコードは破棄され `current` のコードが `old` となるとともに，破棄されたコードを参照しているプロセスは強制的に実行終了される [86]．本実装では，開発者によるコードの変更が確実に適用されるよう，連続して 2 回の適用を行っている．また，本報告の執筆時点においては，ファイルの更新時刻や内容を考慮しない実装となっているため，開発者が開発の対象とするファイルすべて（具体的には，開発者が直接の開発対象とする `lib/` ディレクトリ以下に置かれる Elixir のコードを含むファイル）を更新対象とする実装になっている．そのため，本実装によるコードの適用後には，開発対象となるコードによって生成された Erlang VM 上のプロセスの再起動が発生する．

なお，実装については GitHub 上のリポジトリ[3]，および，Elixir のライブラリリポジトリである Hex[4]でオープンソースソフトウェア（OSS）として公開している．

[3] `https://github.com/kentaro/mix_tasks_upload_hotswap`
[4] `https://hex.pm/packages/mix_tasks_upload_hotswap`

4.3 実験と評価

本章では，提案方式について実験に基づき既存方式と比較検討することで評価する．はじめに，4.3.1 項で実験の対象と方法について述べる．次に，4.3.2 項で既存方式と提案方式について実験した結果を示す．最後に，4.3.3 項で評価結果についての発展的な議論を述べる．

4.3.1 対象と方法

本章における実験は，2.2 節で検討した既存方式と 4.2 節で検討した提案方式とを対象に行う．それぞれ以下の通りであり，(3) が提案方式である．

1. ファームウェアイメージの全体を適用する方式
2. ファームウェアイメージの差分を適用する方式
3. アプリケーションコードを動的に適用する方式

(1) および (2) については，IoT アプリケーションの実装に用いた Nerves が提供する機能を用いてコードの変更を適用する．(1) については `mix firmware` コマンドを，(2) については `mix firmware.patch` コマンドを用いてファームウェアイメージを生成し，`mix update` コマンドを用いてデバイスへの配備を行う．(3) については，4.2.2 項で述べた実装を用いる．

それぞれの方式を比較するための指標として，開発者によるコードの変更をデバイスに適用した上で，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要する時間を用いる．更新対象の IoT アプリケーションとして，TCP ソケット経由でクライアントから受け取った文字列をそのまま返却するサーバ（いわゆる Echo サーバ）を実装し，IoT デバイス上で動作させる．コードを更新した際，(1) および (2) についてはデバイスの再起動に加えて IoT アプリケーションの起動が発生し，その間は IoT アプリケーションはもとよりデバイスへの通信も遮断される．，一方で，(3) については，前述の通り IoT アプリケーションが動作する Erlang VM 上のプロセスの再起動が発生するが，デバイスの再起動は発生しないためデバイスへの通信に関しては影響がない．しかしその場合でも，IoT アプリケーションが TCP 接続を要する実装である場合，更新の間は対象アプリケーショ

表 4.2: 方式（1）および（2）における更新のフェーズと計測方法

Phase	Measurement Methods
1. Firmware Build (full/patch)	`time` command
2. Firmware Deployment	`time` command
3. Device Reboot	`ping` command
4. Application Reboot	`ncat` command

表 4.3: 方式（3）における更新のフェーズと計測方法

Phase	Measurement Methods
1. Code Deployment	`time` command
2. Application Reboot	`ncat` command

ンとの通信が遮断される．そのため，コードの変更を適用した後に再び IoT アプリケーションと TCP 通信が確立することをもって，いずれの方式においても動作が再開したとみなすことが，評価の条件をそろえるという観点で妥当である．そのため，IoT アプリケーションのレベルで通信するオーバーヘッドの少ない機能として Echo サーバを実装し，実験に用いる．

評価指標とした，開発者によるコードの変更をデバイスに適用した上で，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要する時間は，方式（1）および（2）については**表 4.2** の通り，方式（3）については**表 4.3** の通り内訳を分解できる．また，それぞれのフェーズにかかる時間を計測する方法についても記載した．また，実験に用いた環境は，開発用のホストおよび IoT デバイスについて，それぞれ**表 4.4** の通りである．

4.3.2 実験結果

4.3.1 項で述べた対象と方法に基づき，既存方式と提案方式について，開発者によるコードの変更がデバイスに適用され，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要する時間を計測した．その結果は，**表 4.5** の通りである．また，各方式における更新対象となったファイルサイズを**表 4.6** に示した．

開発者によるコードの変更を IoT デバイスに適用し，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要した時間は，方式（1）では合計で 66.88

表 4.4: 実験環境

	Item	Specification
Dev Host	Model	MacBook Pro 13-inch, 2018
	CPU	2.7 GHz Quad-Core Intel Core i7
	Memory	16 GB 2133 MHz LPDDR3
IoT Device	Model	Raspberry Pi 3 Model B
	Network	Wired LAN
	Platform	Nerves 1.7.2

表 4.5: 各方式によるコードの更新に要した時間の比較（単位：秒）

Method	Firmware Build	Firmware Deployment	Code Deployment	Device Reboot	Application Reboot	Total	Ratio*
(1) Firmware Update (full)	29.45	14.49	—	22.51	0.63	66.88	100
(2) Firmware Update (patch)	30.09	17.46	—	22.54	0.54	70.63	105
(3) Proposed Method	—	—	3.30	—	N/A	**3.30**	**5**

* Ratio column shows the ratios when the total time of the method (1) is set to 100.

秒，方式（2）では 70.63 秒であったのに対し，提案方式である（3）では 3.30 秒であり，提案方式が既存方式より短かった．方式（1）および（2）はファームウェアイメージの生成，デバイスへの適用，IoT デバイスの再起動のそれぞれにおいて時間を要しており，提案方式はそれらの時間を要する処理を必要としないため優位であることが確かめられた．方式（2）は更新対象のファイルサイズが（1）の 7 分の 1 であるにも関わらず，ファームウェアイメージの差分の生成および適用に方式（1）よりも時間を要している．(1）と違って差分を生成すること，および，差分に基づく適用を実行することにオーバーヘッドが存在するからであると考えられる．また，方式（3）に関する Application Reboot については，計測できなかった．更新対象の Echo サーバは更新の実行時に新しいコードを読み込んで再起動するが，本実験によっては起動中の通信が遮断された状態を捉えられない速度で起動が完了したためである．

なお，実験のプロトコルおよび結果に関する詳細は GitHub 上のリポジトリ[5]で公開している．

[5] https://github.com/kentaro/ipsj-sigse207-paper-experiments

表 4.6: 各方式における更新対象ファイルのサイズ

Method	Size
(1) Firmware Update (full)	42MB
(2) Firmware Update (patch)	6MB
(3) Proposed Method	12KB

4.3.3 議論

前項で示した実験結果により，提案方式が開発者によるコードの変更を適用する速度を向上させることで，IoT アプリケーションの開発効率の向上が見込めることを確認できた．一方で，提案方式の実装については議論の余地が残されている．実験に用いた提案方式の実装は，変更対象のコードの依存関係を考慮することなくコードの動的な置き換えを行うため，更新対象のコード間に複雑な依存関係がある場合は，不整合を起こす可能性があると考えられる．また，本実装では IoT アプリケーションが依存する外部モジュールの追加・更新・削除には対応していないため，依存モジュールに関するコードの変更は適用されない．ただし，不整合が起きたり依存モジュールに関する変更があったりした場合でも，ファームウェアイメージを更新することで容易に対応できる．また，プロトタイピング時や開発時のように小規模な変更を頻繁に適用する状況において，提案方式によって開発効率を向上できることは実験結果から確かであるため，提案方式の有効性を損なう問題ではないと考える．

提案方式の一般性について確認する．提案方式は，本章で用いたものとは異なる言語によっても実装可能であると考えられる．本章は IoT アプリケーションの開発効率の向上を目的としているため，プロトタイピング時や開発時における利便のために，相対的にスペックの高い開発プラットフォームを利用可能であることが前提である．そのため，ネイティブコードを直接実行する場合にくらべ相対的にリソースを多く要求する Erlang VM のような仮想機械による実行環境を利用可能である．よって，同様に他の仮想機械を前提とした実行基盤を持つ言語を用いても，提案方式について実装が可能であると考えられる．また，Erlang VM を用いる言語自体の IoT デバイスにおける利用可能性についても，プロトタイピング時や開発時にとどまらない適用が可能である見込みがある．Nerves の開発元では

実運用を前提とした開発を行っており，実際に利用事例も報告されている [87]．また，[72] は AtomVM という IoT デバイス上で動作することに特化した Erlang VM を開発しており，[37] が Middle-end と分類する ESP8266 の後継である ESP32 上で動作する．よって，今後，提案方式が配備後のデバイスに対しても適用できる可能性もあると期待できる．

4.4 本章のまとめ

本章は，IoT アプリケーションの開発効率の向上という観点から，開発者によるコードの変更をデバイスへ適用する方式を提案し実装した．先行研究に基づき方式を整理した上で，提案方式と既存方式とを実験により比較検討した．そのた結果として，開発者によるコードの変更を IoT デバイスに適用し，IoT アプリケーションが変更を取り込んだ状態で動作するまでに要した時間について，提案方式は 95%の改善を実現できた．IoT アプリケーションのプロトタイピング時や開発時においては小規模な変更を頻繁に適用することから，提案方式が実現した改善結果は IoT アプリケーションの開発効率の向上に大きく寄与するものと考える．このことから，IoT デバイスの開発プラットフォームにとって，提案方式が示した IoT アプリケーションを構成するコードの動的な書き換えをサポートすることは，競争優位性となり得ると考えられる．その意味においても，本章で述べた提案手法の意義を評価できると考える．

第5章　構成的統合性の導入

本章では，WebAssembly を用いた動的かつ同型な IoT システムを提案・実装し，IoT システムの開発に対して構成的統合性を導入する．

5.1　はじめに

IoT（Internet of Things）は，センサーやアクチュエーターを備えたスマートデバイスの普及により，リアルタイムのデータ収集と制御を可能にし，私たちの生活や事業の方法を変革している．IoT デバイスは，スマートホーム，ウェアラブルデバイス，スマートシティなど，さまざまな用途で使用されている．これらの IoT デバイスは膨大な量のデータを生成し，その分析によってさまざまな問題の解決や新たなビジネスチャンスの創出につながっている．さらに，高速データ通信と低遅延通信を可能にする 5G ネットワークの導入により，さらなる発展が期待されている．

一方で，IoT デバイスの急速な普及により，開発上の課題が生じている．開発者は，多様かつ急速に変化するユーザーのニーズに応えるため，IoT デバイスを迅速かつ頻繁に開発・更新する必要がある．IoT デバイスを迅速に更新するためには，その機能を動的に変更できなければならない．さらに，クラウドコンピューティングやエッジコンピューティングの発展に伴い，IoT アプリケーションはデバイスとしてだけでなく，異なるプラットフォームやアーキテクチャで構成された階層型システムとしても構築されている [8]．加えて，IoT デバイスは異なるハードウェアで構成される場合がある．したがって，このような IoT システムの開発において，異なるプラットフォームやアーキテクチャで構成された層で異なる技術を使用することは，効率性と保守性の問題につながる．

本章では，前述の問題を解決するために，WebAssembly [18]（Wasm）を用いた

2つの方法を提案する．第一に，Wasmを使用してデバイスを再起動せずにIoTデバイスの動作をオンデマンドで部分的に更新できる動的IoTアプリケーションを提示する．提案手法では，IoTデバイスは主要言語とWasm実装で構成され，Wasm実装部分を動的に更新することで動的IoTアプリケーションを構築できる．これにより，開発者は多様かつ急速に変化するユーザー要求に迅速かつ頻繁に対応することが可能となる．第二に，IoTシステムの層間で共通のコードベースから同一のWasmバイナリを使用する同型IoTシステムを提案する．これは，層間で実行されるコードの同型性を表現している．提案手法では，Wasmの移植性を活用することで，異なるプラットフォームやアーキテクチャで構成されうるIoTシステム全体で同一のWasmバイナリを使用できる．共通のコードベースを使用することで，IoTシステム開発の複雑さを低減できる．

動的IoTアプリケーションを構築するために，ハードウェアとしてRaspberry Pi 4 [88]を使用し，Elixirプログラミング言語 [52]でIoTデバイスを開発するためのプラットフォームであるNerves [53]を用いてIoTデバイスを実装する．さらに，IoTデバイスの主要な実装言語であるElixir上でWasmランタイムを実行する．Wasmランタイム上で実行されるWasmモジュールとElixir間の処理を中継するためにWasmtube [89]を開発する．これにより，ElixirコードがWasmモジュール内の関数を呼び出して処理を実行し，その結果をIoTデバイスの動作に反映させることが可能となる．さらに，WasmtubeはWasmバイナリの置き換えを検出し，新しいバイナリに基づいてWasmランタイムを再起動する．これにより，デバイス全体を再起動することなくIoTデバイスを動的に更新することが可能となる．

同型IoTシステムのユースケースとして，機械学習モデルをWasmバイナリにコンパイルし，共通のコードベースから構築された同一のWasmバイナリが複数の層で動作することを実証する．まず，Apache TVM [90]を使用して，ONNX [91]形式で配布され画像認識と分類を行うことができる機械学習モデルをWasmバイナリにコンパイルする．その後，このWasmバイナリをIoTシステムの複数の層の各アプリケーションにデプロイする．各アプリケーションは，Elixir上でWasmランタイムを実行するWasmtubeを使用して実装される．同一のWasmバイナリによって実現される機械学習モデルは，そのWasmランタイム上で動作する．Wasmモジュールは，画像内容を引数としてElixirアプリケーションから実行され，画像

認識と分類処理を行う．この同型アーキテクチャにより，機械学習モデルを使用した推論処理の実用的なユースケースにおいて，プラットフォームやデバイス間で共通のコードベースを実現できる．さらに，各層が Wasm に基づく共通のコードベースを持つことで，ワークロードを考慮した処理のオフローディング [92] が容易になる．

提案手法の有効性を示すために，2 つの観点から評価を行う．まず，IoT デバイスが Elixir による主要言語と Wasm ランタイム上で動作する Wasm モジュールで構成される場合に，本手法によって導入されるオーバーヘッドを測定する．実験結果から，アプリケーションから Wasm 関数を呼び出す際に，アプリケーション内で処理が完了する場合と比較して約 200μs のオーバーヘッドが発生することが確認された．しかし，この時間は IoT デバイス全体の処理時間と比較して短い．さらに，Wasm モジュール自体の処理時間が長くなるほど，相対的にオーバーヘッドは小さくなる．このオーバーヘッドは十分に小さく，提案手法の有効性を示すのに許容できる範囲である．次に，ResNet-50 [93] および MobileNetV2 [94] からコンパイルされた機械学習モデルを Wasm バイナリに適用することで，IoT システムの層間で同一の Wasm バイナリを使用する実現可能性を確認する．IoT デバイス，クラウドサーバー，およびローカルマシンで画像認識と分類を実行した．最も性能の低い Raspberry Pi 4 上でも，Wasm ランタイム上で動作する MobileNetV2 が約 0.6s で実行されることを確認した．将来的に Wasm が GPU などのハードウェアアクセラレーションを容易に活用できるようになれば，提案手法の有効性はさらに向上すると考えられる．

5.2 提案手法と実装

提案手法を 5.2.1 項で，その実装を 5.2.2 項で説明する．

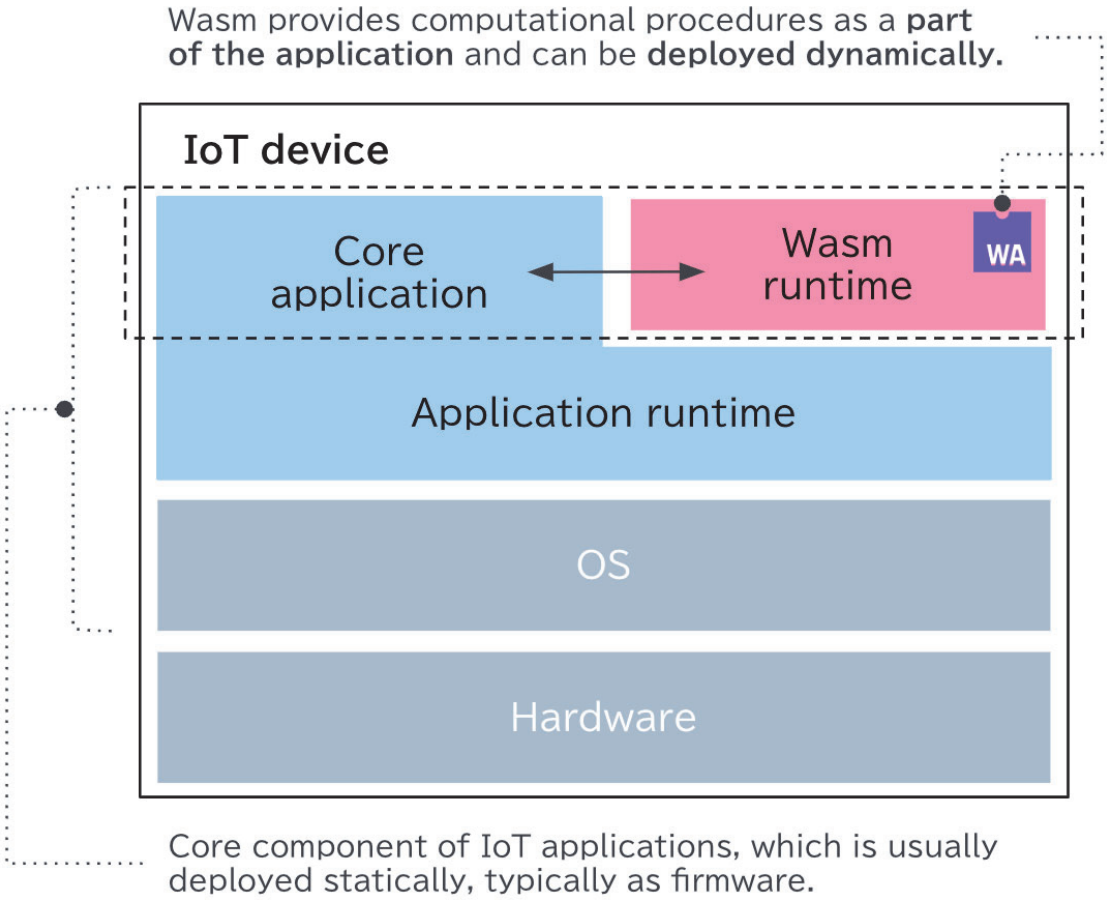

図 5.1: 動的 IoT アプリケーションのための提案手法の概要

5.2.1 提案手法

動的 IoT アプリケーションのための Wasm

まず，開発者が IoT デバイスを動的に更新することを可能にする Wasm ベースの手法を提案する．**図 5.1** に，提案手法の概要を示す．提案手法は，ハードウェア上でオペレーティングシステム，アプリケーションランタイム，およびアプリケーションが動作する一般的な構成の IoT デバイスを想定している．これらは通常，IoT デバイスを実装する単一のファームウェアとして構築される．提案手法では，IoT デバイスを実装するアプリケーションは，コアアプリケーション部分と Wasm ランタイム上で動作する部分の 2 つに分割される．Wasm によって実装される部分は，専用の Wasm ランタイムによって実行される．

Wasm は WASI [95] と呼ばれるシステムコール抽象化層を含んでいる．しかし

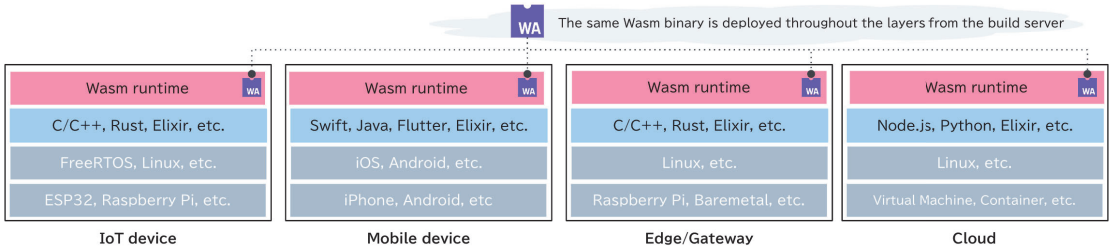

図 5.2: 同型 IoT システムのための提案手法の概要

ながら，WASI は現時点では，さまざまな IoT デバイスを網羅するのに十分な汎用性を備えていない [39]．そこで，提案手法では，アプリケーションのコア部分がシステムインターフェースを介して下位層へのアクセスを担い，その後，Wasm ランタイムと統合される構成をとる．これにより，アプリケーション全体を更新・再起動することなく，Wasm バイナリを置き換えることでアプリケーションの動作を動的に更新できるようになるとともに，汎用的な IoT デバイスの実装が可能となる．開発者は，提案手法を用いることで，多様かつ急速に変化するユーザー要求に迅速に対応することができる．

同型 IoT システムのための Wasm

さらに，IoT システムの複数の層にわたって共通のコードベースを使用できるようにする Wasm ベースの手法を提案する．**図 5.2** に，提案手法の概要を示す．提案手法は，上述の動的に更新可能な IoT デバイス方式と同様に，IoT システムの各層のアプリケーションが Wasm ランタイムを含むことを前提としている．前述のように，Wasm はプラットフォームに依存しない実行形式である．この事実を活用することで，同一の Wasm バイナリを異なるプラットフォームやデバイスの任意の層にデプロイして実行することができる．提案手法では，共通のコードベースから構築された同一の Wasm バイナリが各層にデプロイされる．

提案手法では，各層がアプリケーション実行環境の一部として独自の Wasm ランタイムを持つだけでなく，共通のコードベースから構築された同一の Wasm バイナリをデプロイして実行することで，コードの同型性を実現する．これにより，異なるプラットフォームおよびデバイス間で共通のコードベースを使用することが可能となり，結果として，IoT システム開発の効率性と保守性の向上が期待で

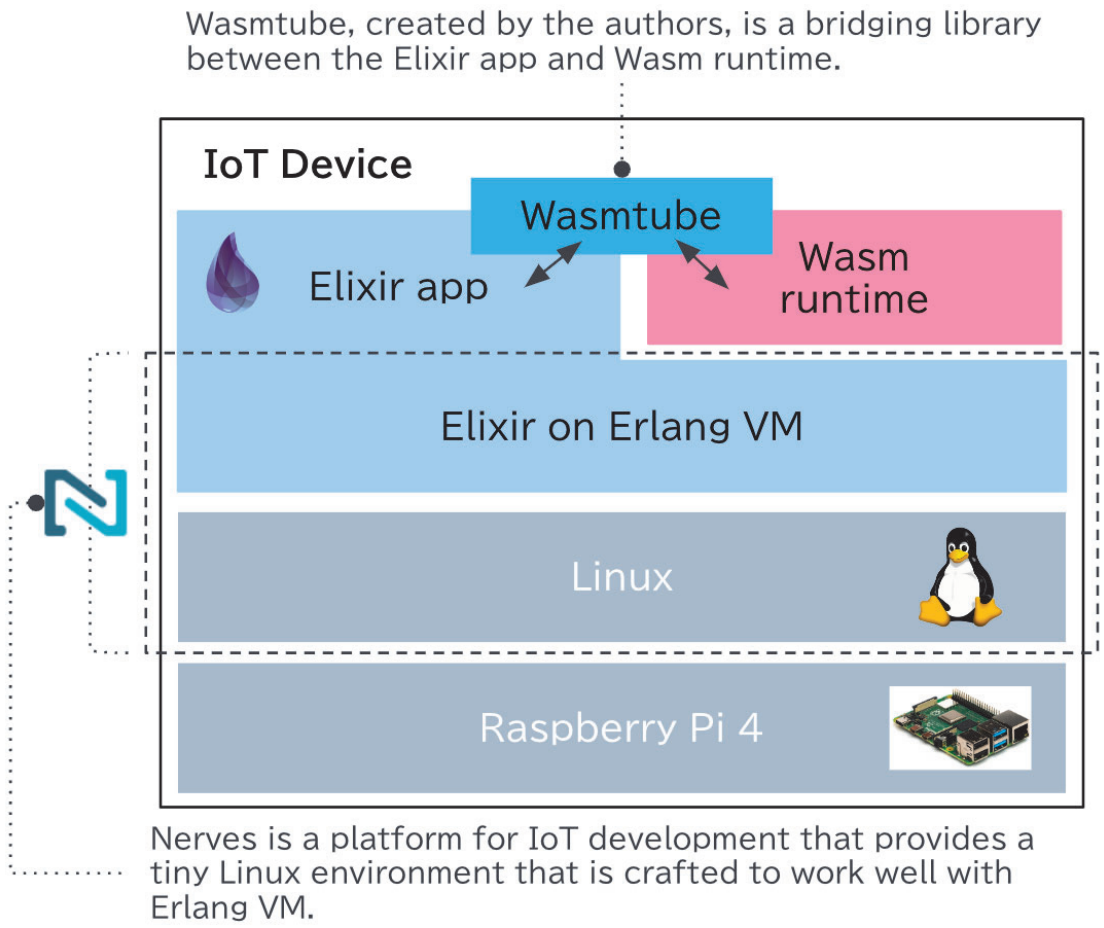

図 5.3: 動的 IoT アプリケーションのための提案手法の実装

きる．

5.2.2 実装

動的 IoT アプリケーションのための Wasm

提案する動的 IoT アプリケーションを構築するために，**図 5.3** に示すように提案手法を実装した．IoT デバイスの実装には，Elixir プログラミング言語 [52] に基づく IoT 開発プラットフォームである Nerves [53] と，ハードウェアとして Raspberry Pi 4 [88] を使用した．Wasm ランタイムは，IoT デバイスを実装するための主要言語である Elixir 上で動作する．これは，IoT デバイスにおいて動的に更新可能なプロセスを実装する Wasm モジュールを実行する．Wasm ランタイム上で動作する Wasm モジュールと Elixir との間の処理を中継するために Wasmtube [89] を開発

した．これにより，Elixir コードから Wasm モジュール内の関数を呼び出すことができる．その結果，Elixir で動作する IoT デバイスは Wasm モジュールからの結果を反映できるようになる．さらに，Wasmtube は Wasm バイナリが置き換えられたことを検出すると，新しいバイナリに基づいて Wasm ランタイムを再起動することで，IoT デバイスを動的に更新することができる．

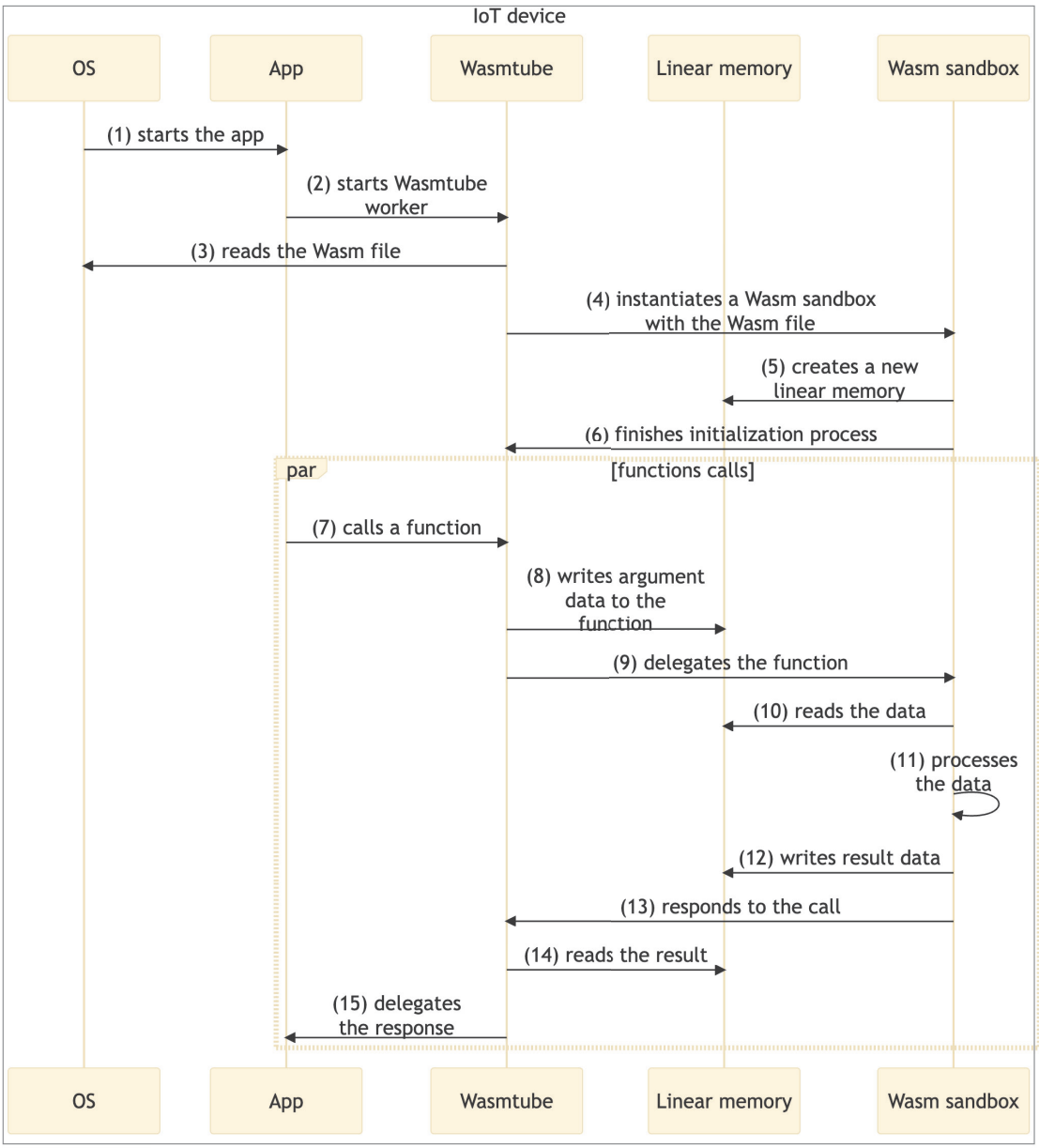

図 5.4: Wasmtube の動作のシーケンス図

提案手法における Wasmtube の動作について，2 段階に分けて説明する．まず，

図 5.4 に Elixir アプリケーション内で Wasmtube が Wasm を初期化し，関数呼び出しを転送する方法を示す．アプリケーションは Wasmtube を起動し，Wasmtube は指定された Wasm バイナリを読み込む（**図 5.4** (1)–(3)）．Wasm ランタイムは，指定された Wasm バイナリを実行するための隔離された環境であるサンドボックスと，専用の線形メモリ領域を作成する．初期化後，アプリケーションが Wasm サンドボックスにアクセスするためのインターフェースが提供される（**図 5.4** (4)–(6)）．アプリケーションは Wasmtube を介して Wasm が提供する関数を呼び出す．Wasmtube は，呼び出された関数の引数をバイナリ列として Wasm サンドボックス内の線形メモリに書き込む．Wasmtube は，引数が JSON エンコードされた文字列にエンコードできる Elixir データ型，または画像バイナリとそのサイズとして渡されることを期待する．その後，Wasmtube はアプリケーションの指定された Wasm 関数を呼び出す（**図 5.4** (7)–(9)）．Wasm 関数はまず，線形メモリから引数データを読み込む．このデータに基づいて処理を行い，結果を線形メモリに書き込む．関数が実行を完了すると，プロセスは Wasmtube に戻り，Wasmtube は Wasm 処理の結果を線形メモリから読み取り，アプリケーションに返す（**図 5.4** (10)–(15)）．アプリケーションはこの結果に基づいて処理を継続する．

図 5.5 は，アプリケーションにおいて，Wasmtube が Wasm を再初期化し，新たにデプロイされた Wasm バイナリに基づいて IoT デバイスの動作を更新する方法を示している．開発者は，IoT デバイスの動作を更新するために，新しい Wasm バイナリをオペレーティングシステムにデプロイする（**図 5.5** (1)）．Wasm バイナリが更新されると，Linux の `inotify` API はポーリングを行うことなく Wasmtube に更新を通知する．Wasmtube は，通知に基づいて更新された Wasm バイナリを再読み込みする（**図 5.5** (2)–(3)）．Wasmtube は，既存の Wasm サンドボックスを破棄し，更新された Wasm バイナリに基づいて新しいサンドボックスを初期化する（**図 5.5** (4)–(5)）．初期化プロセスとそれに続く関数呼び出しは，**図 5.4** と同様である．図では IoT デバイスの外部からの Wasm バイナリのデプロイを示しているが，IoT デバイスがネットワーク経由で新しい Wasm バイナリを取得することも可能である．いずれの方法で実現するにせよ，Wasm バイナリを置き換えることで，IoT デバイスは動的にその動作を更新できるようになる．

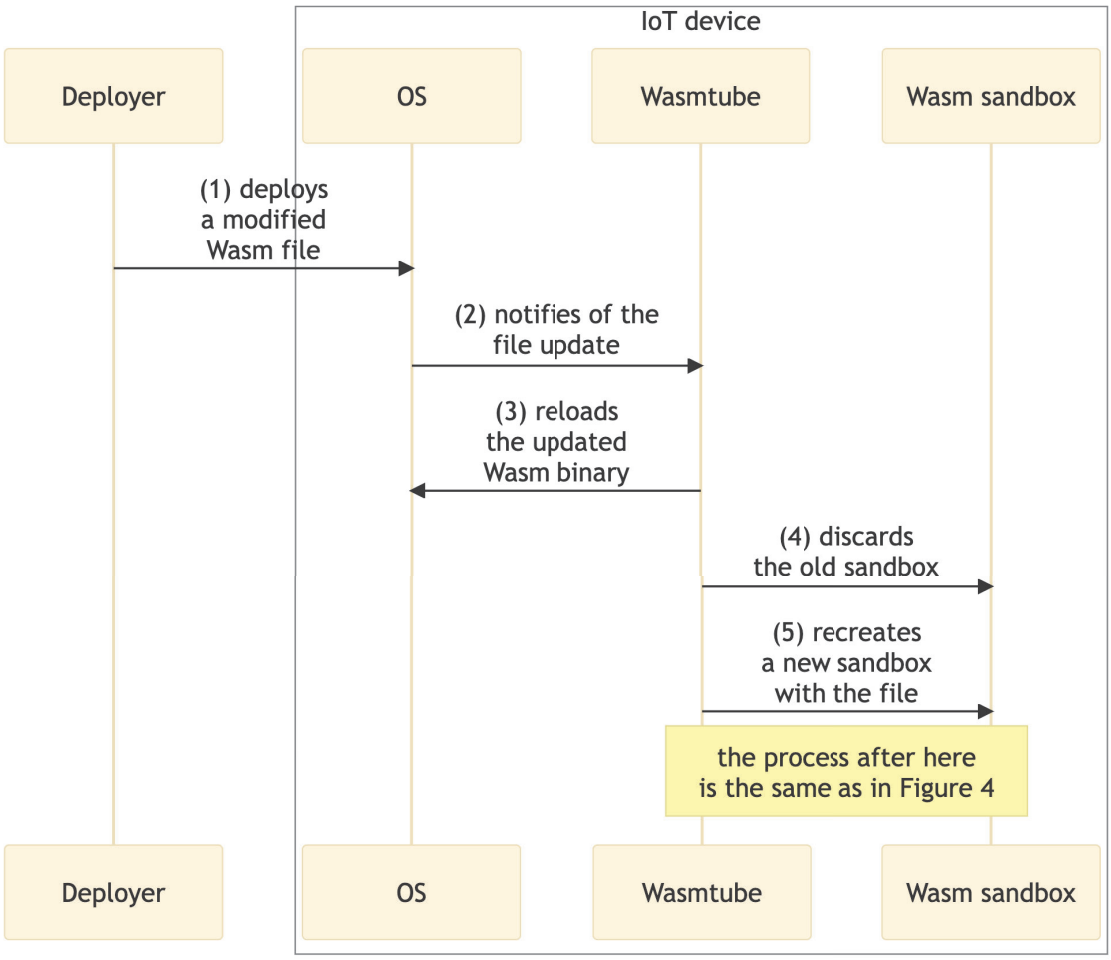

図 5.5: Wasmtube がファイル更新を処理するシーケンス図

Wasm を用いた同型 IoT システム

同型 IoT システムのユースケースを示すため，機械学習モデルを用いて画像認識と画像分類を行う IoT システムを具体例として選択した．この実装では，前述の Wasmtube を用いて実装したアプリケーション上で画像認識と画像分類タスクを実行し，機械学習モデルは同一の Wasm バイナリにコンパイルした．実装の概要を**図 5.6** に示す．

機械学習モデルを Wasm バイナリにコンパイルする手順について説明する．ONNX [91] フォーマット[1]で配布されている画像認識および画像分類モデルは，Apache TVM [90]（以下，TVM と呼ぶ）を用いて単一の Wasm バイナリにコンパイルさ

[1]onnx/models: ONNX フォーマットの事前学習済み最先端モデル集 `https://github.com/onnx/models`

図 5.6: 同型 IoT システムのための提案手法のユースケース

Prediction with Wasm binary (ResNet-50)

```
resnet50_wasm = Wasmtube.from_file("/data/wasm/resnet50.wasm")

resnet50_result =
  resnet50_wasm
  |> Wasmtube.call_function("predict", image: kino_image.content, width: 224, height: 224)
```

Evaluated

```
%{
  "data" => [-1.2380532, -3.5076628, -1.745974, -2.5777662, -2.575174, 0.00320144, -1.8548435,
   1.9798119, 0.8790855, 2.6920712, -1.6338683, -2.262256, -1.7829849, -3.352644, -2.6420937,
   -1.8500642, -1.9090271, -0.8531388, -0.61099184, -3.1587193, -2.0874774, 2.5533094, 1.2311407,
   1.499493, 2.5900023, -1.0595945, -1.6094693, -1.8300521, -1.3385713, -2.9864905, -0.78965205,
   -2.2536402, -1.1391659, -2.6450782, -3.3188202, -3.0392742, -2.900531, -2.6652336, -0.8872474,
   -0.7320179, -2.1677976, -2.5951917, -1.1870259, 0.06935866, -1.2533929, -0.8644694, -3.7154303,
   0.23462734, -0.81006914, ...],
  "dtype" => "FP32",
  "shape" => [1, 1000],
  "strides" => nil
}
```

図 5.7: Elixir アプリケーションから Wasm への関数呼び出しの例

れる．TVM は，機械学習モデルを，さまざまな実行環境に適したさまざまな形式にコンパイルするためのフレームワークである．TVM は Wasm へのコンパイルもサポートしているため，機械学習モデルを単一の Wasm バイナリにコンパイルすることができる．まず，TVM が提供するスクリプトを使用して，ONNX フォーマットで記述されたファイルから機械学習モデルとパラメータを抽出した．次に，Rust プログラミング言語を用いて，TVM が提供するライブラリを使用してモデルとパラメータを読み込み，引数として渡された画像に対して認識と分類を実行する関数を実装した．その後，Rust で実装された関数を Wasm バイナリにコンパイルした．上記の方法を用いて画像認識および画像分類モデルを Wasm バイナリにコンパイルするために使用したコードは，GitHub リポジトリ[2]で公開されている．

[2] https://github.com/kentaro/wasm-standalone-builder

生成されたWasmバイナリは，IoTシステムを構成する各層のオペレーティングシステムにデプロイされる．各層でElixirを実行するアプリケーションとしては，ブラウザ上でElixirコードを実行可能なLivebook [96] を用いた．各アプリケーションにおいて，前述のようにWasmランタイムはWasmtubeを用いてElixirアプリケーション上で動作する．Wasmによって提供される関数は，Wasmtubeを介してElixirアプリケーションから呼び出され，画像の内容が引数として渡される．アプリケーションは，Wasmによる処理結果に基づいて処理を継続する．**図 5.7**は，上述のように実装されたIoTデバイス上のアプリケーションによる画像認識と画像分類タスクの実行結果を示している．同様に，同一のWasmバイナリによって実現された機械学習モデルは，他の層のアプリケーションによっても実行された．提案手法により，異なるプラットフォームおよびデバイス間で共通のコードベースを使用して，機械学習モデルによる推論処理を実装することが可能になる．

5.2.3 提案手法を実装する上での技術的課題

提案手法の要となる筆者が開発したWasmtubeの実装においては，解決するべき技術的に困難な課題があった．まず，ElixirとWasmランタイムとの連携における技術的課題を解決する必要があった．関連研究においては整数，および，浮動小数点数のみにしか対応していなかったコアアプリケーションとWasmで実装された処理間のデータ受け渡しについて，Wasmのサポートする線形メモリを活用することで，画像のようなより複雑なデータ構造を受け渡し可能な方法を実装した．また，Wasmバイナリの動的な更新を実現するために，ファイルシステムのイベントを非同期的に監視し，変更検知時にランタイムを再初期化する機構を構築した．さらに，Elixirの並行処理モデルとWasmランタイムの実行モデルの差異を調整し，競合状態を回避するための同期機構を実装した．これらの実装には，Elixir上で動作するようWasmランタイムを組み込むためのプログラミング技術，および，Wasmランタイムの技術的詳細についての知識が必要であった．これらの技術的課題を克服することで，IoTアプリケーションをWasmとの組み合わせとして構成可能なWasmtubeの実装を実現した．

表 5.1: 実験環境

Machine	CPU	Architecture
Raspberry Pi 4 Model B	Cortex-A72 1.5 GHz	ARMv8-A
Memory	OS	
4GB	Linux (Nerves)	

5.3 評価

提案手法の評価について，5.3.1 項では動的 IoT アプリケーション，5.3.2 項では同型 IoT システムを対象としてそれぞれ示す．

5.3.1 動的 IoT アプリケーションのための Wasm

提案手法では，IoT デバイスの動的な更新を可能にするために，アプリケーションはコアアプリケーションと Wasm を組み合わせた構成をとる．5.2.2 項で説明した提案手法の実装に用いた Nerves は，ファームウェアの更新方法を提供している [97]．この方法では，アプリケーションの更新を完了するために，オペレーティングシステム全体を再起動する必要がある．**表 5.1** に示した実験環境では，予備実験により，OS の再起動とアプリケーションサービスが利用可能になるまでに 10 秒以上かかることが確認された．一方，提案手法では Wasm ランタイムの再起動のみが必要であり，後述する実験で使用した機械学習モデルの Wasm バイナリの再読み込みとランタイムの再起動には約 1.4 秒かかる．これは，オペレーティングシステムの再起動を必要としない提案手法の方がはるかに高速であることを意味する．しかし，アプリケーションと Wasm 間の関数呼び出しのオーバーヘッドが必然的に発生する．提案手法によってもたらされるオーバーヘッドの量を特定し，その有効性を検証するために実験を行い，その汎用性について議論する．

実験方法

提案手法によってもたらされるオーバーヘッドを計測するため，アプリケーションを構成する主要言語のみを用いた場合と Wasm ランタイムを用いた場合におけ

表 5.2: 性能比較実験の結果

Name	IPS*	Average	Deviation	Median	99th%
Elixir	32.24K	31.01μs	± 65.48%	31.17μs	43.26μs
Wasm	3.96K	252.48μs	± 22.50%	240.26μs	399.59μs
Wasm (Noop)	4.05K	246.67μs	± 15.16%	233.78μs	392.25μs

*Iterations per second.

る IoT デバイスの実装の差異を調査した．実験環境を**表 5.1** に示す．実験設定として，1 秒ごとにセンサデータを取得し，1 分ごとに統計処理を行い，ネットワークを介して上位層にデータを送信する IoT デバイスを想定した．オーバーヘッド計測のワークロードとして，60 個の整数型数値の配列を引数として平均値と中央値を計算する関数を用いた．この計算を，Elixir で実装されたアプリケーションのみで実行した場合と，Wasm で実装された関数呼び出しによって実行した場合を比較した．さらに，Wasm 関数呼び出し自体のオーバーヘッドを確認するため，前述の計算を行わずに固定値を返すだけの Wasm 関数を計測した．この実験を通して，提案手法によってもたらされるオーバーヘッドを確認することができた．

実験結果

表 5.2 は，上記の方法[3]を用いて実施した実験の結果を示している．Elixir のみで実装されたアプリケーション（**表 5.2** の Elixir の行）で上記の計算を実行した場合，平均時間は 31.01 μs，中央値は 31.17 μs であった．Wasm で実装された計算を関数呼び出しによって実行した場合（**表 5.2** の Wasm の行），平均時間は 252.48 μs，中央値は 240.26 μs であった．計算を行わずに固定値を返すだけの Wasm 関数を呼び出した場合（**表 5.2** の Wasm (Noop) の行），平均は 246.67 μs，中央値は 233.78 μs であった．このことから，アプリケーションを構成する主要言語のみで計算処理を行う方が，Wasm で実装された関数を呼び出して計算処理を行うよりも高速であることがわかる．Wasm の関数呼び出しの結果を比較すると，その差はわずかである．したがって，オーバーヘッドは Elixir と Wasm の速度差ではなく，Elixir と Wasm の関数呼び出しのオーバーヘッドによるものであることがわかる．

[3]実験の詳細については `https://github.com/kentaro/wfiot2023` を参照されたい．

表 5.3: 実験環境

Layer	Machine	CPU	Architecture	Memory	OS
Device	Raspberry Pi 4 Model B	Cortex-A72 1.5 GHz	ARMv8-A	4GB	Linux (Nerves)
Cloud	Google Cloud Compute Engine e2-medium	2 vCPU (Intel Broadwell)	x86/64	4GB	Debian GNU/Linux 11
Local	iMac	Apple M1	arm64	16GB	macOS 13.2

議論

前述の実験により，提案手法によって導入されるオーバーヘッドが確認された．このオーバーヘッドは，アプリケーションを実装する主要言語である Elixir から Wasm 関数を呼び出す処理に依存するものであり，Elixir と Wasm ランタイム間の性能差に起因するものではない．したがって，実行されるタスクによらず，オーバーヘッドはほぼ一定値であると仮定できる．すなわち，Wasm で処理されるタスクが複雑になり，処理時間が長くなるにつれて，総処理時間に対するオーバーヘッドの割合は減少する．ゆえに，IoT デバイスの性能要件が約 200 μs のオーバーヘッドを許容できる場合，Wasm を用いて IoT デバイスを動的に更新するという提案手法は十分に有効であると言える．

ブラウザ内部における Wasm の性能を評価した研究では，Wasm ランタイム上でコードを実行した場合，ネイティブコードと比較して約 1.5 倍の性能低下が報告されている [98]．本章のように，Wasm がスタンドアロンランタイム上で実行された場合の性能比較に関する報告が今後期待される [39]．著者らが開発した Wasmtube は，よく知られた Wasm ランタイムである Wasmtime [99] を，Wasmex [100] と呼ばれる Elixir 拡張ライブラリを介して使用している．しかし，[101] で概説されているように，現在多くの Wasm ランタイムが利用可能であり，将来的には IoT デバイスに特化した，より性能の高い Wasm ランタイムが登場すると考えられる．

提案手法の汎用性について議論する．提案手法の実装には，IoT デバイス向けのハイエンドなハードウェアファミリに分類される Raspberry Pi 4 を用いている [37]．[40] では，[37] によってミッドレンジに分類される ESP32 上で動作するコードを Wasm3 [102] と統合することで，動的に更新可能な IoT デバイスを構築している．しかし，この研究で提示された実装は，Wasm 関数呼び出しの過程において整数と浮動小数点数のサポートに限定されている．そのため，本章の提案のように機械学習モデルを用いた推論処理を実行するには適用が難しいと考えら

表 5.4: 機械学習モデルを用いた性能比較実験の結果

Layer	Model	IPS*	Average	Deviation	Median	99th%
Device	ResNet-50	0.30	3304.31 ms	± 0.14%	3303.16 ms	3319.68 ms
	MobileNetV2	1.70	588.69 ms	± 0.22%	588.92 ms	590.93 ms
Cloud	ResNet-50	0.89	1120.75 ms	± 20.77%	1065.05 ms	2224.26 ms
	MobileNetV2	5.93	168.55 ms	± 1.31%	168.19 ms	178.07 ms
Local	ResNet-50	3.39	295.32 ms	± 9.49%	286.29 ms	472.07 ms
	MobileNetV2	17.12	58.41 ms	± 18.37%	56.75 ms	100.44 ms

*Iterations per second.

れる．これは，本章で述べる成果がより広範な応用可能性を持つことを示唆している．

5.3.2 同型 IoT システムにおける Wasm

提案手法は，共通のコードベースを用いながら，異なるプラットフォームおよびデバイスからなる多層 IoT システム上で機械学習モデルによる推論処理を実行するという，実用的なユースケースに基づいて実装した．実装したアプリケーションにおいて，これらの機械学習モデルを用いた推論処理の実行速度を測定した．本節では，提案手法の現在の実現可能性と将来展望について議論する．

実験方法

5.2.2 目で概説したように，Wasm バイナリにコンパイルされた機械学習モデルを用いて，多層 IoT システムで一般的に使用される複数の環境において，推論処理の性能を測定した．**表 5.3** に実験環境を示す．クラウド層としては，Google Cloud Compute Engine によって提供される e2-medium インスタンスを使用した．この広く使用されているインスタンスタイプは,「低コストで日常的なコンピューティング」を提供するとされている[4]．比較のために，ローカルの iMac でも実験を行った．画像認識と分類のための機械学習モデルとして，ResNet-50 [93] と MobileNetV2 [94] を使用した．ResNet-50 は，画像認識と分類のために広く使用されているモデル

[4] https://cloud.google.com/compute/docs/machine-resource

である．しかし，これは約 2,500 万個のパラメータを持つ大規模なモデルであり，通常は GPU 上で実行される．そのため，実験では，モバイルフォンなどの比較的低消費電力のデバイスでも高い性能を実現する MobileNetV2 も使用した．実験では，これらの機械学習モデルを Wasm にコンパイルし，Wasm ランタイム上で実行した．その際，推論処理と性能測定のために，Elixir アプリケーションから Wasmtube を介して Wasm に画像が渡された．

実験結果

上述した手法を用いて実験を行った．まず，ResNet-50 と MobileNetV2 の両方が，すべての層において推論タスクを正しく実行できることを確認した．具体的には，画像をアップロードし，Wasm にコンパイルされたモデルを用いて画像の推論を行い，正しい結果が得られることを確認した．**表 5.4** は，アプリケーションから Wasm 関数を呼び出すことによって得られた性能の結果を示している[5]．MobileNetV2 はすべての層において ResNet-50 よりも優れた性能を示し，ResNet-50 の平均実行時間はデバイス層とクラウド層で 1s を超えたのに対し，MobileNetV2 の平均実行時間は，最も効率の悪いデバイス層でも 588.69ms であった．

議論

前述の実験では，IoT システムの各層において画像認識と分類モデルを実行する Wasm にコンパイルされた機械学習モデルを用いて，実際に推論処理を実行できることを示した．その結果，デバイス層における推論処理は ResNet-50 では 3s 以上を要するのに対し，MobileNetV2 では約 0.6s で済むことが確認された．したがって，機械学習モデルとして MobileNetV2 を採用した場合，約 1FPS の処理速度が許容されるならば，共通のコードベースから構築された同一の Wasm バイナリを用いて同型 IoT システムを構築する提案手法は依然として有効である．

Wasm はポータブルなインタフェースを目指しているため，GPU などのハードウェアアクセラレーションを利用することが難しい．そのため，本実験環境では CPU によって推論処理を実行した．したがって，前述のように，デバイス層にお

[5]実験に関する詳細は `https://github.com/kentaro/wfiot2023` を参照されたい．

いて ResNet-50 のような比較的大規模なモデルを用いて実用的な性能で推論処理を実行することは困難である．機械学習モデルの実行を高速化できる GPU などのハードウェアに対する Wasm からの共通インタフェースを定義するために，wasi-nn [103] と呼ばれる仕様が現在開発されている．このような仕様が IoT デバイスで使用されているハードウェアで利用可能になれば，提案手法の有効性が将来的に向上すると期待される．

さらに，同型性をより徹底した実装を検討することができる．提案手法の同型 IoT システムの実装では，アプリケーションの各層の実装に Elixir を用いた．しかし，5.2.2 目で概説したようにで説明したように，機械学習モデルを Wasm バイナリにコンパイルするために Rust を使用した．機械学習モデルを Elixir を用いて Wasm バイナリとして実装できれば，Wasm のレベルだけでなく，その実装言語のレベルでも同型性を実現できる．Firefly [104] は，Elixir で記述されたアプリケーションを Wasm にコンパイルするコンパイラとして開発されている．将来的には，このコンパイラを用いて，ONNX 形式で配布されている機械学習モデルを Wasm バイナリにコンパイルすることができるようになる可能性がある．これが実現すれば，提案手法の実装における同型の度合いが，Wasm バイナリのレベルだけでなく，その実装言語のレベルでも向上することが期待される．

5.4 本章のまとめ

クラウド/エッジコンピューティングの進化に伴う IoT デバイスおよび IoT システムの普及によって IoT デバイスの開発において生じる問題を解決するために，Wasm の利用を提案した．まず，Wasm を用いて動的に更新可能な IoT デバイスを実現する方法を提案した．次に，IoT システムの異なる層において，共通のコードベースから構築された同一の Wasm バイナリを使用する同型 IoT システムを提案した．これらの提案を実装し，実用的なユースケースにおける適用を実証した．提案手法の現在の実現可能性を確認し，将来展望について議論するために，定量的な評価を行った．

今後の展望として，機械学習モデルを Elixir で実装し Wasm バイナリにコンパイルすることで，Wasm レベルだけでなく実装言語レベルでも同型性を実現する

ことが考えられる．そのことで，より統合的な IoT システムの実現が可能となる．

第6章　結論

本研究では，異種混合的（heterogeneous）な構成が当然視されてきたIoTシステムに対して，開発の複雑さを低減することを目的として動的かつ同型なIoTシステムアーキテクチャ–Dynamic Isomorphic IoT System Architecture–を提案・実装し，その有効性を検証した．本章では，本論文をまとめ結論とする．

6.1　本研究の成果

IoTシステムに対して，開発の複雑さを低減することを目的として統合的アーキテクチャを導入することを提案した本研究の成果について，**図 6.1** に示す．

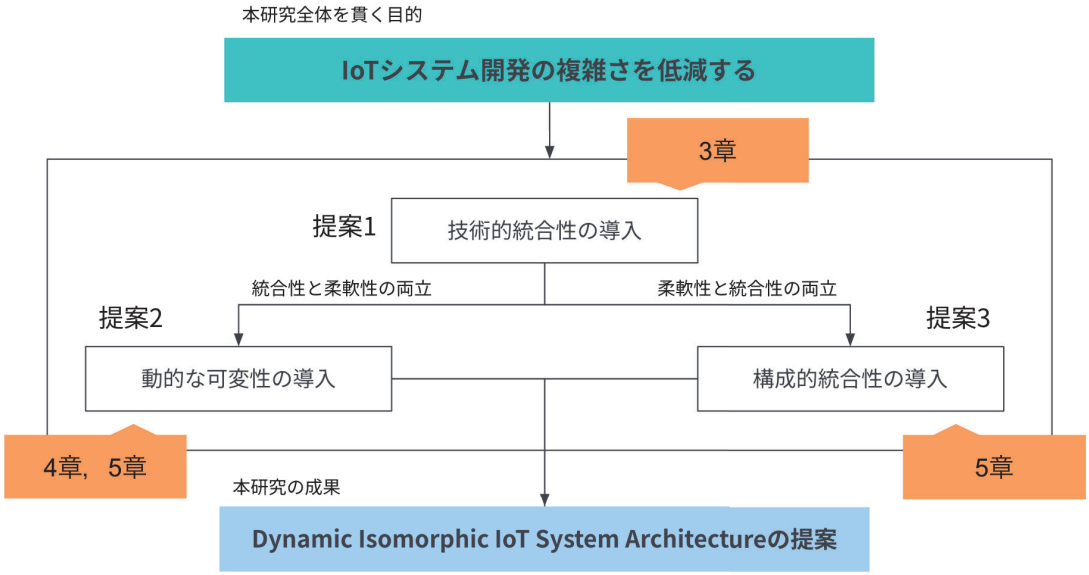

図 6.1: 本研究の成果

第1に，IoTシステム全体を単一のプログラミング言語で統合的に設計・開発するデータフロー基盤を構築した．IoTシステム全体を一貫して開発するための技術基盤の構築を目指して，Elixir言語とそのエコシステムを活用し，デバイス層か

らクラウド層まで一貫した開発環境を実現した．これにより，各層間を統合的に開発可能としつつ，IoT システム内のデータフローを見通しよく記述できる技術基盤を提案した．実装としてデータフロー基盤 Pratipad を開発し，実際のユースケースを通じて有効性と適用可能性を示した．

第 2 に，IoT デバイス内のアプリケーションに対して，開発者がコードの変更を動的に適用できる手法を提案・実装した．統合的なシステムアーキテクチャを効果的に実現しつつ，同時に，システムを状況やニーズへの変化に対して柔軟に対応させることを目指した．提案手法では，実行中のデバイスにコードの変更を即時に適用できるため，迅速な開発サイクルが実現できる．性能評価の結果，既存方式と比較して，更新に要する時間を大幅に短縮できることを実証した．

第 3 に，WebAssembly（Wasm）を活用した動的・同型な IoT システムアーキテクチャを提案・実装した．単一の言語による開発では，新技術の取り込みにおける制約が生じる可能性があるため，ポータブルな仕様に基づく Wasm を用いることで，技術的統合性を維持しつつ必要な機能を柔軟に取り込むことを目指した．Wasm バイナリの置き換えにより IoT デバイス内のアプリケーションの動的な更新を可能とするとともに，各層で共通の Wasm バイナリを使用できる構成的統合性を実現した．画像認識モデルを用いたユースケースを通じて，実用的な範囲で性能を担保しつつ，技術的統合性を可能な限り損なうことなく Wasm による機能追加と更新が可能であることを示した．

以上の提案を総合することで，異種混合的構成が当然視されたきた IoT システムの開発に対して，統合性と柔軟性をかねそなえる統合的アーキテクチャ–Dynamic Isomorphic IoT System Architecture–を実現した．これにより，IoT システム開発における複雑さを低減することができた．本研究の成果は，今後の IoT システム開発分野の発展に貢献するものであり，さまざまな応用分野での活用が期待される．今後の IoT システムは，昨今の発展著しい AI 技術のような大きな変化を次々と取り込んでいくことが必要となり，ハードウェアおよびソフトウェアの両面で技術が進歩することで，デバイス層やエッジ層における AI 活用が進んでいくと考えられる．そのような IoT システムの開発において，迅速な開発・更新が可能となり技術革新への対応力を高めることができる本研究の成果は，特に有用であると考える．

6.2　本研究の一般的な適用可能性

本研究では，Elixir 言語と Erlang 分散ネットワークという具体的な技術を用いて提案手法の実装と検証を行った．そのため，一般的な適用可能性について疑義を呈される可能性がある．特に，Elixir や Erlang が他の主要なプログラミング言語に比べて採用事例が限定的である現状において，本手法が広範なシステムや異なる開発環境で適用可能かどうかが懸念される．

しかしながら，本研究で提案した動的かつ同型な IoT システムアーキテクチャは，特定の言語やプラットフォームに依存しない汎用的なものである．動的にコード変更を適用する方式は，Python，JavaScript などの他のプログラミング言語でも実現可能であり，各言語が持つ動的な読み込みなど機能を活用することで同様の効果を得られる．また，WebAssembly（Wasm）のプラットフォーム非依存性を活用することで，異なるデバイスや環境間でのコード共有と再利用が可能となり，同型性の実現に寄与する．

一方で，IoT システムを構成するデバイスは用途に応じて適切なスペックのものを用いるのが合理的な選択となることは今後も変わらない．そのため，相対的に高性能なハードウェアを前提とする本手法を IoT システム開発のすべての場面に適用することは適切ではないと考える．しかしながら，ハードウェアおよびソフトウェアの両面で技術が進歩し，より小型で高性能なデバイスが登場しそれらを用いて提案手法が実現可能となることで，適用範囲は今後さらに広がると期待される．

本手法の核心は，IoT システム開発における複雑さの低減にあり，これは IoT 分野全体が抱える共通の課題である．したがって，本研究の成果は特定の技術に限定されず，他の多様なプラットフォームや開発環境にも適用可能性を有すると考える．今後は，提案手法のさらなる一般化と，他のプログラミング言語やプラットフォームへの適用可能性を検証することで，IoT システムの持続的な発展と広範なエコシステムの構築に貢献していきたい．

6.3　本研究の社会的な意義

本研究は，IoT システム開発における異種混合的な構成がもたらす開発の複雑さを低減する必要があるという問題に対して，動的かつ同型な IoT システムアーキテクチャを提案・実装した点で社会的意義がある．

まず，IoT 技術は産業，医療，農業，交通など多岐にわたる分野で活用されており，その発展は社会全体の効率化と利便性の向上に直結している．しかし，従来の開発手法では，異なるプラットフォームや言語の混在により，開発コストの増大や保守性の低下が深刻な課題となっていた．本研究の提案するアーキテクチャにより，開発者は単一のコードベースでシステム全体を構築・更新できるため，IoT システム開発の複雑さが低減し，環境の変化への迅速な対応が可能となる．

また，動的なコード適用方式の実現は，セキュリティホールの迅速な修正や新機能の即時展開を可能にし，IoT デバイスの安全性とユーザビリティを高める．これにより，社会インフラや公共サービスの信頼性が向上し，ユーザー体験の質的向上にも寄与する．さらに，WebAssembly を用いた同型性の実現は，異なるプラットフォーム間での相互運用性を高め，新たなサービスやアプリケーションの創出を促進する．これにより，IoT エコシステムの拡大と産業全体の活性化が期待できる．

総じて，本研究は IoT システムの開発・運用における根本的な課題を解決し，持続可能で柔軟性の高い IoT 社会の実現に大きく貢献するものである．

6.4　今後の展望

今後の課題として，提案手法のさらなる一般化と，他のプログラミング言語やプラットフォームへの適用可能性の検討が挙げられる．本研究では主に Elixir 言語と Nerves プラットフォームを用いて実装を行ったが，これらに限定されず，より多様な環境での適用を可能にすることで，提案手法の汎用性を高めることができる．特に，リソースが限られたデバイスや，異なるアーキテクチャを持つシステムでも同様の効果が得られるよう，軽量化や最適化の手法を研究する必要がある．

また，セキュリティやスケーラビリティの面での評価を行い，大規模な IoT システムへの適用性を高めることが求められる．提案手法が多数のデバイスやユー

ザーを含む環境でどのような性能を示すか，また安全性をどのように維持するかを検証することは重要である．特に，動的なコード適用が可能であることによるセキュリティリスクを最小限に抑えるための機構や，負荷分散・冗長化といった大規模システム特有の課題に対する対策を講じる必要がある．

さらには，異なる技術との相互運用性を確保し，より広範なエコシステムを構築することで，IoT システムの持続的な発展に寄与していきたい．他のプラットフォームやプロトコルとの互換性を持たせることで，既存のシステムやサービスとの統合が容易になり，新たな価値の創出につながると考えられる．

謝辞

本研究に取り組むにあたり，主指導教員としてご指導くださった篠田陽一教授に感謝いたします．ゼミ等を通じた研究に対する直接のご指導はもとより，技術にとどまらない幅広い知見に基づくお話をいただくことで，より広い世界への視野を開いていただきました．篠田教授の本学における最後の年度に学位取得が間に合ったことは，幸運の極みです．

副指導教員の宇多仁准教授からは毎回のゼミを通じて研究へのご指導を賜りました．田中清史教授からは副テーマ研究でご指導をいただきました．鈴木正人准教授，高瀬英希准教授（東京大学）は，本論文の審査員をお引き受けくださり，本研究に対して有益なご意見をくださいました．丹康雄教授，山崎進准教授（北九州市立大学）から，4 章の元となる論文においてご指導を賜りました．感謝いたします．また，篠田研究室の皆様，特に小比賀亮仁博士，村本衛一博士から折に触れて温かいご助言をいただきました．ありがとうございます．

GMO ペパボ株式会社の皆様にも感謝いたします．特に，ペパボ研究所の三宅悠介博士，客員研究員の力武健次博士には，共著や研究会でのディスカッション，日々のお話を通じて，さまざまなご尽力・ご助言をいただきました．ペパボ研究所という社内研究所の所長という立場である以上，自らも研究能力を身につけたいという私の思いと取り組みに対して，さまざまな面においてバックアップをしてくださいました．研究開発という面からも事業成長に貢献していけるよう，引き続き精進します．

最後になりますが，妻の桂以子，博士後期課程在学中に生まれた息子の環太郎に感謝いたします．ついつい他のことをそっちのけで研究やプログラミングに熱中して家庭をあまり顧みない不祥の夫をおおらかな心持ちで見守ってくれました．今後ともよろしくお願いします．

ご支援くださった皆様に重ねて御礼を申し上げ，謝辞とさせていただきます．

本研究に関する業績

論文誌（査読あり）

- [19] 栗林健太郎, 三宅悠介, 力武健次, 篠田陽一. Pratipad：IoT システムを単一のプログラミング言語で統合的に構築できるデータフロー基盤の提案. 情報処理学会論文誌, Vol. 64, No. 3, pp. 635–649, March 2023.

国際会議（査読あり）

- [21] Kentaro Kuribayashi, Yusuke Miyake, Kenji Rikitake, Kiyofumi Tanaka, and Yoichi Shinoda. Dynamic IoT applications and isomorphic IoT systems using WebAssembly. In 2023 IEEE 9th World Forum on Internet of Things (WF-IoT), pp. 1–8. IEEE, October 2023.

国内会議（査読あり）

- [105] 栗林健太郎, 三宅悠介, 力武健次, 篠田陽一. IoT システムの双方向データフローにおける設計と実装の複雑さを解消する手法の提案. インターネットと運用技術シンポジウム論文集, Vol. 2021, pp. 48–55, November 2021. （優秀論文賞，優秀プレゼンテーション賞を受賞）

国内研究会

- [20] 栗林健太郎, 山崎進, 力武健次, 丹康雄. IoT デバイス内アプリケーションの開発効率向上のためにコードの変更を動的に適用する方式の提案と実

装. 情報処理学会・研究報告ソフトウェア工学（SE）, Vol. 2021-SE-207, No. 32, pp.1–8, February 2021.

国外技術会議

- [106] Kentaro Kuribayashi. Pratipad: A declarative framework for describing bidirectional dataflow in iot systems with elixir. ElixirConf 2021, October 2021.

技術解説書

- [107] 栗林健太郎, 大原常徳, 大聖寺谷一樹, 山内修, 齋藤和也, 隆藤唯章, 高瀬英希. Elixir 実践入門 : 基本文法，Web 開発，機械学習，IoT. Web+DB Press プラスシリーズ. 技術評論社, 2024.

参考文献

[1] Shashi Kant Gupta, Radha Raman Chandan, Rupesh Shukla, Prabhdeep Singh, Ashish Kumar Pandey, and Amit Kumar Jaiswal. Heterogeneity issues in IoT-driven devices and services. Journal of Autonomous Intelligence, Vol. 6, No. 2, p. 588, aug 1 2023.

[2] Muhammad Noaman, Muhammad Sohail Khan, Muhammad Faisal Abrar, Sikandar Ali, Atif Alvi, and Muhammad Asif Saleem. Challenges in Integration of Heterogeneous Internet of Things. Scientific Programming, Vol. 2022, pp. 1–14, aug 16 2022.

[3] Tie Qiu, Ning Chen, Keqiu Li, Mohammed Atiquzzaman, and Wenbing Zhao. How Can Heterogeneous Internet of Things Build Our Future: A Survey. IEEE Communications Surveys & Tutorials, Vol. 20, No. 3, pp. 2011–2027, 2018.

[4] Euihyun Jung, IlKwon Cho, and Sun Moo Kang. An Agent Modeling for Overcoming the Heterogeneity in the IoT with Design Patterns. Lecture Notes in Electrical Engineering, pp. 69–74, 2014.

[5] Mehdi Mahmoodpour, Andrei Lobov, and Minna Lanz. Configurator module to integrate different protocols for IoT solution. In 2018 IEEE 16th International Conference on Industrial Informatics (INDIN). IEEE, 7 2018.

[6] Chaitali Parmar, Atharva Todankar, Shrihari Wayal, and Prof. Jagat Gaydhane. Middleware to Address Heterogeneity Problem in IoT. International Journal of Advanced Research in Science, Communication and Technology, pp. 306–312, apr 23 2022.

[7] Deepak Choudhary. Security Challenges and Countermeasures for the Heterogeneity of IoT Applications. Journal of Autonomous Intelligence, Vol. 1, No. 2, p. 16, jan 31 2019.

[8] Sharu Bansal and Dilip Kumar. IoT Ecosystem: A Survey on Devices, Gateways, Operating Systems, Middleware and Communication. Int. J. Wireless Inf. Networks, Vol. 27, No. 3, pp. 340–364, September 2020.

[9] Alae-Eddine Bouaouad, A. Cherradi, S. Assoul, and N. Souissi. Architectures and emerging trends in Internet of Things and Cloud computing: a literature review. 2020 4th International Conference on Advanced Systems and Emergent Technologies (IC_ASET), 2020.

[10] Eisha Akanksha, Aritri Debnath, and Barnali Dey. Extensive Review of Cloud Based Internet of Things Architecture and Current Trends. In 2021 6th International Conference on Inventive Computation Technologies (ICICT), pp. 1–9. ieeexplore.ieee.org, January 2021.

[11] Cristian Martin, Daniel Garrido, Manuel Diaz, and Bartolome Rubio. From the Edge to the Cloud: Enabling Reliable IoT Applications. In 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, 8 2019.

[12] K. Yelamarthi, Md Sayedul Aman, and A. Abdelgawad. An Application-Driven Modular IoT Architecture. Wireless Communications and Mobile Computing, 2017.

[13] S. Narasimha Swamy and Solomon Raju Kota. An Empirical Study on System Level Aspects of Internet of Things (IoT). IEEE Access, 2020.

[14] Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems, pp. 10–11. Pearson, Upper Saddle River, NJ, 4 edition, March 2014.

[15] Java. `https://java.com/`. Accessed: 2025-01-02.

[16] React Native · Learn once, write anywhere. https://reactnative.dev/. Accessed: 2025-01-02.

[17] Flutter - Build apps for any screen. https://flutter.dev/. Accessed: 2025-01-02.

[18] WebAssembly. https://webassembly.org/. Accessed: 2025-01-02.

[19] 栗林健太郎, 三宅悠介, 力武健次, 篠田陽一. Pratipad：IoT システムを単一のプログラミング言語で統合的に構築できるデータフロー基盤の提案. 情報処理学会論文誌, Vol. 64, No. 3, pp. 635–649, March 2023.

[20] 栗林健太郎, 山崎進, 力武健次, 丹康雄. IoT デバイス内アプリケーションの開発効率向上のためにコードの変更を動的に適用する方式の提案と実装. 情報処理学会・研究報告ソフトウェア工学（SE）, Vol. 2021-SE-207, No. 32, pp. 1–8, February 2021.

[21] Kentaro Kuribayashi, Yusuke Miyake, Kenji Rikitake, Kiyofumi Tanaka, and Yoichi Shinoda. Dynamic IoT applications and isomorphic IoT systems using WebAssembly. In 2023 IEEE 9th World Forum on Internet of Things (WF-IoT), pp. 1–8. IEEE, October 2023.

[22] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. TinyLink 2.0: integrating device, cloud, and client development for IoT applications. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, MobiCom '20, pp. 1–13, New York, NY, USA, April 2020. Association for Computing Machinery.

[23] Y Nakata, M Takai, H Konoura, and M Kinoshita. Cross-Site Edge Framework for Location-Awareness Distributed Edge-Computing Applications. In 2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), pp. 63–68. ieeexplore.ieee.org, August 2020.

[24] Karan Mitra, Rajiv Ranjan, and Christer Åhlund. Context-Aware IoT-Enabled Cyber-Physical Systems: A Vision and Future Directions. Handbook of Integration of Cloud Computing, Cyber Physical Systems and Internet of Things, pp. 1–16, 2020.

[25] Julius Hülsmann, Jonas Traub, and Volker Markl. Demand-based sensor data gathering with multi-query optimization. Proceedings VLDB Endowment, Vol. 13, No. 12, pp. 2801–2804, August 2020.

[26] Dashbit. Broadway: Concurrent and multi-stage data ingestion and data processing with Elixir. `https://github.com/dashbitco/broadway`. Accessed: 2025-01-02.

[27] Wesley M Johnston, J R Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. ACM Computing Surveys, Vol. 36, No. 1, pp. 1–34, 2004.

[28] OpenJS Foundation and Node-RED contributors. Node-RED. `https://nodered.org/`. Accessed: 2025-01-02.

[29] N K Giang, M Blackstock, R Lea, and V C M Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. In 2015 5th International Conference on the Internet of Things (IOT), pp. 155–162, October 2015.

[30] Narges Yousefnezhad, Avleen Malhi, and Kary Främling. Security in product lifecycle of IoT devices: A survey. Journal of Network and Computer Applications, Vol. 171, p. 102779, December 2020.

[31] Stephen Brown and Cormac J Sreenan. Software Updating in Wireless Sensor Networks: A Survey and Lacunae. Journal of Sensor and Actuator Networks, Vol. 2, No. 4, pp. 717–760, November 2013.

[32] J Bauwens, P Ruckebusch, S Giannoulis, I Moerman, and E D Poorter. Over-the-Air Software Updates in the Internet of Things: An Overview of

Key Principles. IEEE Commun. Mag., Vol. 58, No. 2, pp. 35–41, February 2020.

[33] Peter Ruckebusch, Eli De Poorter, Carolina Fortuna, and Ingrid Moerman. GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules. Ad Hoc Networks, Vol. 36, pp. 127–151, January 2016.

[34] Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, Jeroen Hoebeke, and Eli De Poorter. Modelling the energy consumption for over-the-air software updates in LPWAN networks: SigFox, LoRa and IEEE 802.15.4g. Internet of Things, Vol. 3-4, pp. 104–119.

[35] Ondrej Kachman, Marcel Balaz, and Peter Malik. Universal framework for remote firmware updates of low-power devices. Comput. Commun., Vol. 139, pp. 91–102, May 2019.

[36] Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for IoT networks – A survey. September 2020.

[37] M O Ojo, S Giordano, G Procissi, and I N Seitanidis. A Review of Low-End, Middle-End, and High-End Iot Devices. IEEE Access, Vol. 6, pp. 70528–70554, 2018.

[38] Peter Ruckebusch, Spilios Giannoulis, Ingrid Moerman, Jeroen Hoebeke, and Eli De Poorter. Modelling the energy consumption for over-the-air software updates in LPWAN networks: SigFox, LoRa and IEEE 802.15.4g. Internet of Things, Vol. 3-4, pp. 104–119.

[39] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. WebAssembly Modules as Lightweight Containers for Liquid IoT Applications. In Web Engineering, pp. 328–336. Springer International Publishing, 2021.

[40] István Koren. A Standalone WebAssembly Development Environment for the Internet of Things. In Web Engineering, pp. 353–360. Springer International Publishing, 2021.

[41] ESP32 Wi-Fi & Bluetooth MCU | Espressif Systems. https://www.espressif.com/en/products/socs/esp32. Accessed: 2025-01-02.

[42] Tommi Mikkonen, Cesare Pautasso, and Antero Taivalsaari. Isomorphic Internet of Things Architectures With Web Technologies. Computer, Vol. 54, No. 7, pp. 69–78, July 2021.

[43] Antero Taivalsaari, Tommi Mikkonen, and Cesare Pautasso. Towards Seamless IoT Device-Edge-Cloud Continuum:. In Maxim Bakaev, In-Young Ko, Michael Mrissa, Cesare Pautasso, and Abhishek Srivastava, editors, ICWE 2021 Workshops, pp. 82–98, Cham, 2022. Springer International Publishing.

[44] AWS Lambda. https://aws.amazon.com/jp/lambda/. Accessed: 2025-01-02.

[45] AWS IoT Core. https://aws.amazon.com/jp/iot-core/. Accessed: 2025-01-02.

[46] AWS IoT Greengrass. https://aws.amazon.com/jp/greengrass/. Accessed: 2025-01-02.

[47] Azure IoT Edge. https://azure.microsoft.com/ja-jp/products/iot-edge. Accessed: 2025-01-02.

[48] KubeEdge. https://kubeedge.io/. Accessed: 2025-01-02.

[49] Hany F Atlam, Ahmed Alenezi, Abdulrahman Alharthi, Robert J Walters, and Gary B Wills. Integration of Cloud Computing with Internet of Things: Challenges and Open Issues. In 2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social

Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 670–675. ieeexplore.ieee.org, June 2017.

[50] 松木雅幸. Nature Remo やその関連サービスで使われている技術と今後の展望. https://engineering.nature.global/entry/start-blog. Accessed: 2025-01-02.

[51] Timothy Kieras, Junaid Farooq, and Quanyan Zhu. I-SCRAM: A Framework for IoT Supply Chain Risk Analysis and Mitigation Decisions. IEEE Access, Vol. 9, pp. 29827–29840, 2021.

[52] The Elixir Team. The Elixir programming language. https://elixir-lang.org/. Accessed: 2025-01-02.

[53] The Nerves Project Authors. The Nerves Project. https://www.nerves-project.org/. Accessed: 2025-01-02.

[54] Kuribayashi Kentaro. Pratipad: A Declarative Framework for Describing Bidirectional Dataflow in IoT Systems with Elixir. https://github.com/kentaro/pratipad. Accessed: 2025-01-02.

[55] Kuribayashi Kentaro. Pratipad: A Declarative Framework for Describing Bidirectional Dataflow in IoT Systems with Elixir. https://speakerdeck.com/kentaro/pratipad-a-declarative-framework-for-describing-bidirectional-dataflow-in-iot-systems-with-elixir. Accessed: 2025-01-02.

[56] phoenixframework.org. Phoenix framework. https://phoenixframework.org/. Accessed: 2025-01-02.

[57] Ericsson AB. Erlang Programming Language. https://erlang.org/. Accessed: 2025-01-02.

[58] MicroPython - Python for microcontrollers. https://micropython.org/. Accessed: 2025-01-02.

[59] Espruino - JavaScript for microcontrollers. https://www.espruino.com/. Accessed: 2025-01-02.

[60] TinyGo home. https://tinygo.org/. Accessed: 2025-01-02.

[61] Ericsson AB. Erlang – Distributed Erlang. http://erlang.org/doc/reference_manual/distributed.html. Accessed: 2025-01-02.

[62] The Raspberry Pi Foundation. Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org/. Accessed: 2025-01-02.

[63] The BeagleBoard.org Foundation. BeagleBoard.org - community supported open hardware computers for making. https://beagleboard.org/. Accessed: 2025-01-02.

[64] Igor Kopestenski and Peter Van Roy. Erlang as an enabling technology for resilient general-purpose applications on edge IoT networks. In Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang 2019, pp. 1–12, New York, NY, USA, August 2019. Association for Computing Machinery.

[65] The Elixir Team. Node — Elixir v1.14.0. https://hexdocs.pm/elixir/Node.html#connect/1. Accessed: 2025-01-02.

[66] Ericsson AB. Erlang – global. https://www.erlang.org/doc/man/global.html#register_name-2. Accessed: 2025-01-02.

[67] 栗林健太郎. 通信経路が片方向の場合の Elixir のノード間通信の挙動を確認する. https://zenn.dev/kentarok/articles/9a9e578d5a71b7. Accessed: 2025-01-02.

[68] Kuribayashi Kentaro. pratipad_client. https://github.com/kentaro/pratipad_client. Accessed: 2025-01-02.

[69] Kuribayashi Kentaro. off_broadway_otp_distribution: An OTP distribution connector for Broadway. https://github.com/kentaro/off_broadway_otp_distribution. Accessed: 2025-01-02.

[70] Ericsson AB. Erlang – Using TLS for Erlang Distribution. https://www.erlang.org/doc/apps/ssl/ssl_distribution.html. Accessed: 2025-01-02.

[71] The Nerves Project Developers. Targets — nerves v1.7.16. https://hexdocs.pm/nerves/targets.html. Accessed: 2025-01-02.

[72] Davide Bettio. atomvm/AtomVM: Tiny Erlang VM. https://github.com/bettio/AtomVM. Accessed: 2025-01-02.

[73] Ericsson AB. Erlang – Distribution Protocol. https://www.erlang.org/doc/apps/erts/erl_dist_protocol.html. Accessed: 2025-01-02.

[74] Taras Halturin. ergo: an actor based Framework for creating microservices using technologies and design patterns of Erlang/OTP in Golang. https://github.com/ergo-services/ergo. Accessed: 2025-01-02.

[75] IHS Markit. The Internet of Things: a movement, not a market. https://cdn.ihs.com/www/pdf/IoT_ebook.pdf, 2017. Accessed: 2025-01-02.

[76] Development Boards — Espressif Systems. https://www.espressif.com/en/products/devkits. Accessed: 2025-01-02.

[77] Arduino. https://www.arduino.cc/. Accessed: 2025-01-02.

[78] BeagleBoard.org - community supported open hardware computers for making. https://beagleboard.org/. Accessed: 2025-01-02.

[79] NVIDIA embedded systems for next-gen autonomous machines. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed: 2025-01-02.

[80] Software — Arduino. https://www.arduino.cc/en/software. Accessed: 2025-01-02.

[81] ESP-IDF Programming Guide - ESP32 - ESP-IDF Programming Guide latest documentation. https://docs.espressif.com/projects/esp-idf/. Accessed: 2025-01-02.

[82] Real-Time Bidding Platform — SSP, DSP Solutions - Platform.IO. https://platform.io/. Accessed: 2025-01-02.

[83] Lee, Edward Ashford and Seshia, Sanjit A. Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition. MIT Press, 2017.

[84] Oracle VM VirtualBox. https://www.virtualbox.org/. Accessed: 2025-01-02.

[85] Empowering App Development for Developers — Docker. https://www.docker.com/. Accessed: 2025-01-02.

[86] Chapter 14: Compilation and Code Loading. https://erlang.org/doc/reference_manual/code_loading.html. Accessed: 2025-01-02.

[87] Case Studies. https://www.nerves-project.org/case-studies. Accessed: 2025-01-02.

[88] The Raspberry Pi Foundation. Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org/. Accessed: 2025-01-02.

[89] Kentaro Kuribayashi. Wasmtube: A bridging library which allows you to communicate between Elixir and Wasm. https://github.com/kentaro/wasmtube. Accessed: 2025-01-02.

[90] Apache TVM. https://tvm.apache.org/. Accessed: 2025-01-02.

[91] ONNX. https://onnx.ai/. Accessed: 2025-01-02.

[92] Bo Wang, Changhai Wang, Wanwei Huang, Ying Song, and Xiaoyun Qin. A Survey and Taxonomy on Task Offloading for Edge-Cloud Computing. IEEE Access, Vol. 8, pp. 186080–186101, 2020.

[93] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778, June 2016.

[94] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4510–4520, June 2018.

[95] WASI — The WebAssembly System Interface. `https://wasi.dev/`. Accessed: 2025-01-02.

[96] Home - livebook.Dev. `https://livebook.dev/`. Accessed: 2025-01-02.

[97] The Nerves Project Authors. ssh_subsystem_fwup: Erlang SSH Subsystem for Nerves firmware updates. `https://github.com/nerves-project/ssh_subsystem_fwup`. Accessed: 2025-01-02.

[98] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In USENIX Annual Technical Conference, pp. 107–120, 2019.

[99] Wasmtime. `https://wasmtime.dev/`. Accessed: 2025-01-02.

[100] Philipp Tessenow. Wasmex: Execute WebAssembly / WASM from Elixir. `https://github.com/tessi/wasmex`. Accessed: 2025-01-02.

[101] Stephen Akinyemi. Awesome WebAssembly Runtimes: A list of webassemby runtimes. `https://github.com/appcypher/awesome-wasm-runtimes`. Accessed: 2025-01-02.

[102] Wasm3: A fast WebAssembly interpreter, and the most universal WASM runtime. https://github.com/wasm3/wasm3. Accessed: 2025-01-02.

[103] wasi-nn: Neural Network proposal for WASI. https://github.com/WebAssembly/wasi-nn. Accessed: 2025-01-02.

[104] Firefly: An alternative BEAM implementation, designed for WebAssembly. https://github.com/GetFirefly/firefly. Accessed: 2025-01-02.

[105] 栗林健太郎, 三宅悠介, 力武健次, 篠田陽一. IoT システムの双方向データフローにおける設計と実装の複雑さを解消する手法の提案. インターネットと運用技術シンポジウム論文集, Vol. 2021, pp. 48–55, November 2021.

[106] Kentaro Kuribayashi. Pratipad: A declarative framework for describing bidirectional dataflow in IoT systems with elixir. https://speakerdeck.com/kentaro/pratipad-a-declarative-framework-for-describing-bidirectional-dataflow-in-iot-systems-with-elixir. （参照：2022-01-16）.

[107] 栗林健太郎, 大原常徳, 大聖寺谷一樹, 山内修, 齋藤和也, 隆藤唯章, 高瀬英希. Elixir 実践入門 : 基本文法、Web 開発、機械学習、IoT. Web+DB Press プラスシリーズ. 技術評論社, 2024.