「マルチメディア，分散，協調とモバイル
(DICOMO2020)シンポジウム」 令和2年6月

メディア処理技術の適用拡大のための WebAPI フレームワーク開発・運用

舟橋涼一[1] 多田厚子[1] 増井誠生[1] 原田将治[1]
倉木健介[1] 長谷川尚己[1] 田中竜太[1]

概要：映像・音声を含むメディア処理技術が多用途に利用されている一方で，メディア処理のコア技術を直接扱うためには専門的な知識や環境構築が要求される．コア技術をカプセル化し，WebAPI として公開することは，扱いやすさの面で有効な手段となる．メディア処理技術の WebAPI 公開およびコンテナ化を容易に実現するために，各メディア処理の特性を分類・整理しテンプレート化することでインターフェースを共通化し，ひな型コードを自動生成するフレームワークを開発した．開発フレームワークではクラウド上やオンプレミス上の様々な運用環境への配備・開発・運用を素早く回す CI/CD を実現した．本フレームワークの有効性を評価し，開発工数削減率 14.6%，API ごとの運用コスト削減率 79.2%の効果を確認した．

WebAPI Framework Development and Operation for Expanding Applications of Media Processing Technologies

RYOICHI FUNABASHI[1] ATSUKO TADA[1] MOTOO MASUI[1] SHOJI HARADA[1]
KENSUKE KURAKI[1] NAOKI HASEGAWA[1] RYUTA TANAKA[1]

1. はじめに

映像・音声を含むメディア処理技術の利用が業務用途，コンシューマー用途共に広まっている．一方で，メディア処理のコア技術を扱うためには専門的な知識や環境構築が要求され，アプリケーションの開発が困難であるという課題があった．これを解決するためにはアプリケーション開発者が容易に扱える形にメディア処理技術のインターフェースを整え，環境非依存で呼び出せることが重要である．WebAPI の公開は，アプリケーション開発者側の環境構築が必要無く，容易に呼び出し可能となるため，有効な手段となる．さらに WebAPI 提供側でもサーバ環境に依存せずに運用するためには，WebAPI の機能をコンテナ化することが有効である．

一方でメディア処理を WebAPI として公開可能なコンテナとして実装・運用するには以下の課題がある．

1. メディア処理を WebAPI として公開するサーバの開発において，各メディア処理の特性に応じて適切な入出力を定義する必要があり，かつこれらを各メディア処理コアに受け渡すサーバコードの実装が必要となる．このため WebAPI と各メディア処理技術双方の特性を理解した上で設計・実装を進める必要があり，工数が増大する傾向がある．
2. WebAPI として公開するメディア処理技術のアップデートや，要件変化に対応した場合には，運用中のコンテナも随時アップデートが必要となる．時々刻々と変化する要件に対応し続けるため，運用コスト及び展開スピードが課題となる．
3. メディア処理技術で扱う映像や音声はデータ量が大きい傾向があり，かつ人の顔や声といった個人情報を含むデータを扱う場合もあるため，ネットワーク負荷やデータセキュリティを考慮した上でコンテナの配備先を柔軟にコントロールする必要がある．具体的には，大容量データや個人情報を含むデータを直接クラウドにアップロードできない場合があることから，クラウド上やオンプレミス上の様々な運用環境へ配備する必要がある．

これらの課題に対応するために，開発・運用をサポートするフレームワークを開発した．本フレームワークは以下の特長を備える．

- 1 に対応するため，各メディア処理の特性を分類・整理しテンプレート化することでインターフェースを共通化し，ひな型コードを自動生成する機能を

1 （株）富士通研究所
FUJITSU LABORATORIES LTD.

WebAPI 開発者に提供した．これにより，メディア処理技術を容易にコンテナとして実装，及び WebAPI として公開可能となった．

- 2，3 に対応するため，メディア処理技術の開発・運用を，LC4RI[1]・Ansible[*1]により管理する構成とした．これにより，開発・運用を素早く回す CI/CD が可能となり，かつコンテナ配備先の柔軟性も実現した．

本フレームワークを用いることで，開発工数削減効果(コード量削減率 14.6%)，および運用コスト削減効果(サーバ全体の削減率 87.5%，API ごとの削減率 79.2%)が得られ，フレームワークの有効性を確認した．

2. 関連研究

既存の RDB 等の単一データソースを入力とし，標準的な方式で WebAPI として公開する技術は存在し，CData API Server[2]等が代表的である．

一方で，既存のモノリシックなシステムを WebAPI として公開するためにはサービスの適切な分割を含め技術的な課題があり，[3]では課題の整理と解決に向けた検討が進められている．

具体的に既存技術を WebAPI 公開した例として，[4]ではネットワーク異常検知技術を WebAPI 化するにあたり，技術を分析した上で WebAPI 化ラッパを実装し，開発工数を削減する方式が提案されている．

提案するフレームワークにおいて，運用管理に LC4RI を用いている．これはシステム運用管理の実施において，作業手順の説明文書と実際の操作内容を記録し，「実行可能な手順書」として保存可能なツール群である．*“構築・運用作業に必要となる，自然言語による技術情報，実行コード，さまざまな状況で起こり得る複数の実行結果，を，Notebook と呼ばれる 1 つの文書にパッケージし，拡張した JupyterNotebook 上で使用することで，実行可能な手順書を実現している．”* ([1]概要より)

LC4RI の拡張機能として，実行コンテクストを維持しつつ ssh コマンドの連続実行を記述可能な方式が提案されている[5][6].

3. 提案するフレームワーク構成

3.1 フレームワークを用いた WebAPI 開発

メディア処理技術を WebAPI として公開する場合に，以下の観点で分類し，テンプレート化した．

- データ IO (文字列/バイナリなど)
- テータ形式(データ実体/データへの参照値)
- レスポンス提供タイミング(同期/非同期)

WebAPI 開発者はメディア処理技術の特性に応じてこれらのテンプレートを選択し，ひな形コードを自動生成することで，コア部分の開発に集中できる．

テンプレートをベースにコードを自動生成するフレームワークを用いた WebAPI の開発手順を図 1 に示す．

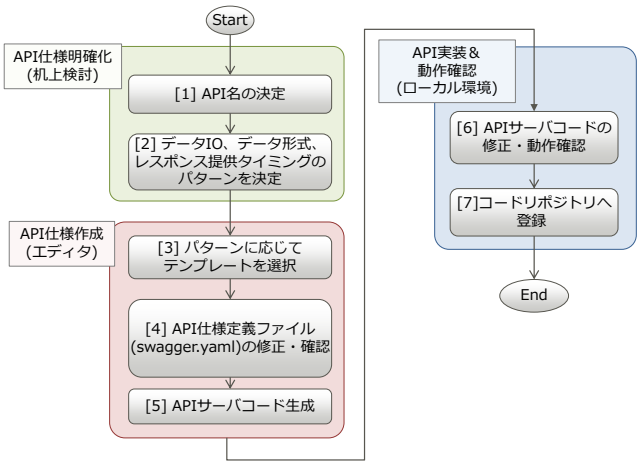

図 1　フレームワークを用いた WebAPI 開発手順

1. API 名の決定
2. データ IO，データ形式，レスポンス提供タイミングのパターンを決定
3. パターンに応じてテンプレートを選択
4. API 仕様定義ファイルの修正・確認
5. API サーバコード生成
6. API サーバコードの修正・動作確認
7. コードリポジトリへ登録

以降，手順の詳細を示す．

3.1.1 API 名の決定

API 名を決定する．処理内容を端的に示す API 名を，英文字 10 字程度で開発者が作成する．各メディア API の URL 形式は最終的に以下の形になる．

https://ドメイン/各メディア処理の API 名/バージョン/API メソッド

3.1.2 データ IO，データ形式，レスポンス提供タイミングのパターンを決定

表 1 は API が用いるデータ IO(リクエスト・レスポンス)のデータ形式，レスポンス提供タイミング(同期・非同期)のパターンから，対応するテンプレートを選択するためのテーブルである．

開発者は API のリクエスト・レスポンスのデータ形式およびレスポンス提供タイミングを，以下より選択する．

*1 https://www.ansible.com/ オープンソースの構成管理ツール

- 文字列: http body 内 json 文字列
- バイナリ: http body 内バイナリデータ
- ファイルの URL: ファイルを参照する URL
  - (レスポンス) 同期出力: リクエストに対するレスポンスとして直接返す
  - (レスポンス) 非同期出力: リクエストを受け取ったのちに ID を発行し，この ID により出力を受け取る
- ストリームの URL: ストリームを参照する URL

これらのパターンから表 1 を参照し，該当するテンプレートを①～⑬より選択する[*2].

表 1 API の各パターンによる選択

レスポンス / リクエスト	文字列 (http body内)	バイナリ(画像) (http body内)	ファイル [同期出力] (画像/文字)	ファイル [非同期出力] (映像/音声)	ストリーム (映像/音声/データ)
文字列 (http body内)	①TT	④TB	⑦TF	–	–
バイナリ(画像) (http body内)	②BT	⑤BB	⑧BF	–	–
ファイルのURL (画像/文字/音声/映像)	③FT	⑥FB	⑨FF	⑩FFa	⑫FS
ストリームのURL (映像/音声)	–	–	–	⑪SFa	⑬SS

映像や音声を扱う場合はデータ量の大きさに応じて以下に留意する.

- データ IO でサイズが大きい場合，ファイルの URL を用いる.
- パラメータなどの指定のみの場合は，文字列を用いる.
- サイズが比較的小さく，かつローカル環境から直接アップロードする必要がある場合は，http body 内バイナリを用いる.

3.1.3 パターンに応じてテンプレートを選択

それぞれのパターンに応じた API 仕様定義ファイルのテンプレートを選択する. テンプレートは OpenAPI と呼ばれる WebAPI の標準仕様に基づく YAML 形式のファイルであり，定義・編集のツールとして Swagger[*3]を用いる.

3.1.4 API 仕様定義ファイル(swagger.yaml)の修正・確認

API 仕様定義ファイルを修正する. Swagger により提供されるエディタを用い，API 仕様のプレビューを見ながら編集できる.

1. basePath にメディア処理の API 名とバージョンを記入
2. “/execute”を適切な処理名(動詞)に修正
3. title や description を処理内容に応じて修正
4. 各メソッドのパラメータ，データ形式を必要に応じて修正

静止画同士の類似度を判定し 0～1 までのスコアで返す画像類似度判定 API の例(抜粋)を以下に示す. パス /match に対して POST でリクエストを出す定義となっている.

```
paths:
  /match:
    post:
      summary: "画像類似度判定実行"
      description: "画像 2 枚の類似度を判定し，類似度の数値を返す¥n"
      operationId: "matchPOST"
      parameters:
      - in: "body"
        name: "body"
        description: "parameters for media process execution."
        required: false
        schema:
          $ref: "#/definitions/Request"
```

3.1.5 API サーバコード生成

Swagger によるサーバコード生成機能を一部カスタマイズした API サーバコード生成により，API 仕様定義ファイルをベースに API サーバコードのスケルトンを生成できる. 画像類似度判定の例では，WebAPI のパス /match への POST リクエストは，以下のコードに変換される. 本フレームワークでは Node.js をサーバコードとして用いているが，他の言語への展開も可能である.

生成されたコード(抜粋)

```
async function matchPOSTInner(args, res, next) {
  // use args.value, return json obj.
  const jsonObj = {
    "score" : 0.0
  };
  return jsonObj;
}
```

3.1.6 API サーバコードの修正・動作確認

スケルトンは WebAPI の各メソッドと対応づけられた空の関数が生成されるため，実際の処理を関数内部(2 行目以降)に実装する.

*2 表中のテンプレート表記(“①TT”など)は以下規約に従う.
{①|…|⑬ (テンプレート番号)}{T|B|F|S (リクエストデータ形式)}{T|B|F|S (レスポンスデータ形式)}[a(非同期出力の場合に付加)]
リクエストデータ形式・レスポンスデータ形式はそれぞれ以下を示す.
T:文字列，B:バイナリ，F:ファイル，S:ストリーム

*3 Swagger は OpenAPI(WebAPI 定義の標準仕様)に準拠したツールを提供する. https://swagger.io/docs/specification/about/

実装コード(抜粋)

```
async function matchPOSTInner(args, res, next) {
  // use args.value, return json obj.
  const inputUrl1 = args.body.value.inputPath1;
  const inputUrl2 = args.body.value.inputPath2;

  //... 中略 ...

  const responseData = fs.readFileSync(outputPath, 'u
tf8');
  const jsonObj = {
    "score" : responseData
  };
  return jsonObj;
}
```

3.1.7 コードリポジトリへ登録

GitHub 等のコードリポジトリへ登録する．この時に，リリースタグとしてバージョン情報を付与することで，運用環境への配備が可能となる．

3.2 LC4RI + Ansible による WebAPI 配備・運用構成

CI/CD 実現に向け，メディア処理技術の WebAPI 開発後の実行サーバへの配備および運用を自動化し，かつスケールアウト可能なサーバ構成へも対応するために，図 2 に示す構成要素を持つフレームワークを構築した．

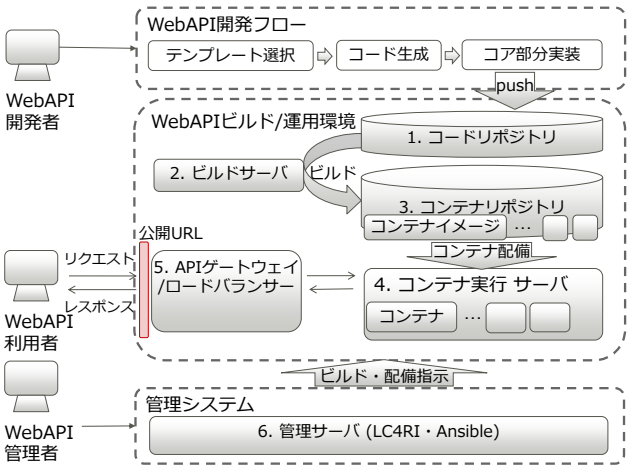

図 2　提案するフレームワーク構成

テンプレート化によりコードの構成も統一がとれた状態のため，“1. コードリポジトリ”に登録された実装コード群を“2. ビルドサーバ”により統一的な手順でコンテナイメージとして自動ビルドし“3. コンテナリポジトリ”に登録できる．これらを各メディア処理の稼働状況およびサーバ数に応じて“4. コンテナ実行サーバ”に配備する．

WebAPI 利用者は“5. API ゲートウェイ/ロードバランサー”を経由してメディア処理を実行する．これらの開発・配備・運用のプロセスは“6. 管理サーバ”により実行され，実行ログが記録される．

WebAPI を新規追加する場合の管理サーバによる運用手順を図 3 に示す．

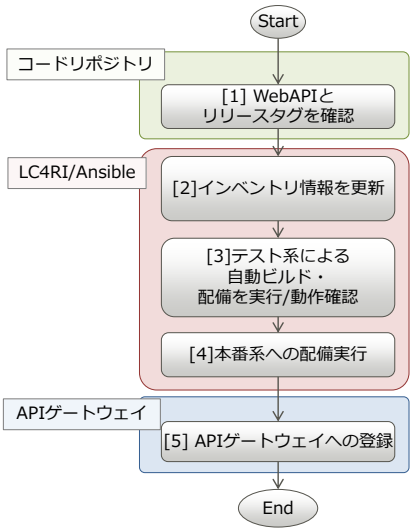

図 3　管理サーバによる運用手順

1. コードリポジトリに登録された WebAPI とリリースタグを確認
2. インベントリ情報を更新
3. テスト系による自動ビルド・配備を実行/動作確認
4. 本番系への配備実行
5. API ゲートウェイへの登録

3.2.1 コードリポジトリに登録された WebAPI とリリースタグを確認

GitHub 等のコードリポジトリへ登録された WebAPI とリリースタグを確認する．

以下の WebAPI を例に手順を示す．

- WebAPI 名: convertImage
- WebAPI の内容: 汎用的な画像の変換を行う
- コードリポジトリ:
  git@github.com:MediaPaaS/WebAPI-convertImage.git
- リリースタグ: 1.0.0

これらの情報に基づき，運用管理者が 3.2.2 以降の操作を行う．

3.2.2 インベントリ情報を更新

以降の手順は LC4RI の playbook 上での操作となる．最初に Ansible のインベントリにおけるパラメータ情報を更新する．

規約に沿った構成でコードリポジトリへ登録されている場合，インベントリ情報は以下パラメータのみを登録すれば以降の処理が実行される．

- WebAPI 名
- リリースタグ
- WebAPI ポート番号

具体的には以下項目を group_vars として追加すればよい．

```
- name: "convertImage"
  git_branch: 1.0.0
  api_port: 9002
```

独自の構成の場合にも，明示的に指定することで配備可能である．
例: Dockerfile，sample ディレクトリが標準構成と異なる位置にある場合の定義

```
- name: "convertImage"
  git_dockerfile_path: v1/docker/alt/convertImage
  sample_src: v1/mnt/convertImage/production/sample
  git_branch: 1.0
  api_port: 9002
```

3.2.3 テスト系による自動ビルド・配備を実行/動作確認

本フレームワークはテスト系として以下を用意している．

1. VM によるテスト系 ：Vagrant ベースの VM 構成により，本番系のサーバ構成を疑似的に作成した環境
2. ステージング系 ： 本番系と同一構成で，ゲートウェイ経由の動作を含めた最終確認を実施可能

新規 API の配備時は，1，2 の順で動作確認を行い，問題が無い場合に本番系への配備へ進む．

運用管理者は LC4RI 上の各テスト環境に対応した notebook を実行する．各 notebook では，それぞれの環境におけるサーバ設定を含む Ansible の定義ファイルを参照し，ビルド・配備を実行する Ansible コマンドを発行する．この操作により，図 2 で示したコンテナビルドから各環境への配備までが自動実行される．またこれらの操作は LC4RI 上にログとして残り，管理される．

動作確認も同様に LC4RI 上で全 API に対してサンプルの動作を自動確認するスクリプトが含まれるため，これを実行する．

3.2.4 本番系への配備実行

テスト系による動作確認が完了した後，本番系への配備を実行する．運用管理者の操作はテスト時と同様であり，動作確認済のコンテナが本番系へ配備される．

3.2.5 API ゲートウェイへの登録

本フレームワークでは Apigee[*4]ベースの API ゲートウェイを用いており，各 API の登録に必要となる API ゲートウェイ用の定義ファイルを自動生成する．

運用管理者は定義ファイルを API ゲートウェイでインポートし，更新された WebAPI の動作を確認する．

4. 実装と評価

4.1 WebAPI 開発と評価

テンプレートをベースにコードを自動生成するフレームワークを用いた WebAPI 開発には以下の利点がある．

1. コード生成機能により，API 開発時の開発量が削減できる
2. WebAPI のパターン別テンプレートの活用により，API 開発時のフローが明確になる
3. WebAPI の形式を揃えることにより，API 利用者にとって統一感のある API 群を提供できる

今回は 1 の開発量削減効果について，コード開発量ベースで測定した．

4.1.1 コード開発量削減効果の測定手順

コード量削減効果を，以下の手順により測定した．

1. コードリポジトリへ登録済の WebAPI から測定対象のコードをサンプリング
2. 測定対象コードの API 仕様定義ファイルから，API サーバコード生成を再実行
3. 再生成されたサーバコードと，実装済コードを比較し，差分ステップ数を計測
4. コード量削減率を 1 - 差分ステップ数/実装済コードのステップ数 により算出[*5]

4.1.2 測定結果

8 つの WebAPI を測定対象としてサンプリングし，測定手順に従いコード量削減率を計測した．
各 WebAPI におけるコード量削減率を図 4 に示す．

*4 標準的な WebAPI 管理機能を備えるゲートウェイ
https://docs.apigee.com/
*5 削減効果は，“本来すべて実装するべきだったところが，差分のみで済んだ”，とみなし算出した．

コード量比率 = 差分コード/実装済コード
コード量削減率 = 1 - コード量比率

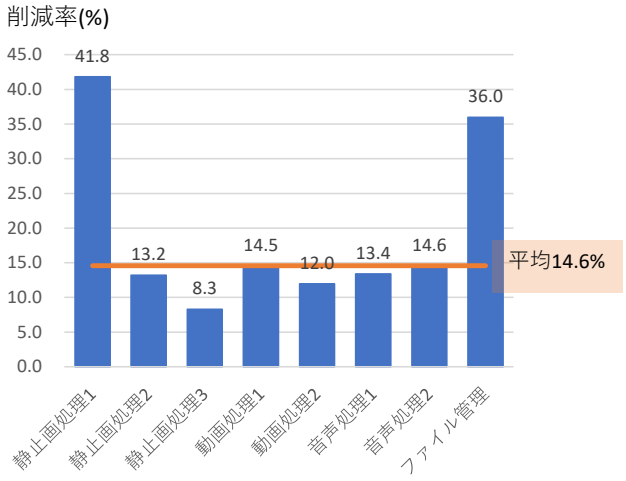

図 4　コード量削減率

平均約 14.6%のコード量削減効果を確認し，開発量削減の効果が示された．

複雑な内部処理が必要な API は固有の事前チェック等が増えるため，削減率は低くなる傾向がある(静止画処理 3 など)．一方で，処理自体が比較的単純な API は生成コードを利用することで大きな削減効果が得られている(静止画処理 1 など)．

また，生成コードで API サーバの大枠が生成され，API 開発者は基本的にメディア処理エンジンの呼び出し部分のみを開発すればいい点も，開発効率の面で効果があると考えられる．

4.2 フレームワークによる配備・運用コストの削減効果

フレームワークによる配備・運用コストの削減効果を，以下の手順により測定した．

1. フレームワーク導入前の設定手順について，サーバ全体の手番数と，API ごとの手番数をカウント
2. フレームワーク導入後の設定手順について同様にカウント
3. 前後の手番数比較により，フレームワークによる配備・運用コスト削減効果を測定

4.2.1 フレームワーク導入前の運用環境

フレームワーク導入前は，設定ファイルのバッチ生成等の自動化は実施していたが，基本的には手動で規約に沿ったディレクトリ構成に開発コードを配備する構成だった．テンプレートは導入済のため，規約自体はある程度シンプルである．同一の API に対する比較も可能であることから，手番の比較対象として用いた．

4.2.2 測定結果

フレームワーク導入前後の測定結果をそれぞれ表 2，表 3 に示す．

表 2 フレームワーク導入前の設定手順

設定先	設定対象	操作手順	サーバ全体の手順	APIごとの手順
バックエンドサーバ	サーバ全体の設定	各種ライブラリインストール	21	
	各APIの設定	APIの展開		3
		テスト		3
		サービス自動化設定		3
リバースプロキシ	サーバ全体の設定	Nginx インストール	3	
	各APIの設定	設定ファイルの編集		5
APIゲートウェイ	各APIの設定	GUIから各APIについて設定		10
合計			24	24

表 3 フレームワークを用いた設定手順

設定先	設定対象	操作手順	サーバ全体の手順	APIごとの手順
LC4RIによる操作	サーバ全体の設定	ssh接続用初期設定	1	
		LC4RIからの接続確認	1	
		サーバアドレス情報の修正	1	
	全APIの設定	API情報の修正		1
		デプロイの実行		1
		動作確認 (全API実行スクリプト実行)		1
APIゲートウェイ	各APIの設定	API情報のファイル読み込み		1
		設定更新実行		1
合計			3	5

削減効果は開発側と同様に，1 – 手番数比率 として算出した．算出結果を以下に示す．

- サーバ全体の手番数削減効果: (1-3/24) * 100 = 87.5%
- API ごとの手番数削減効果: (1-5/24) * 100 = 79.2%

特に API ごとの削減効果は API 数に比例し増大するため，フレームワーク導入による効果は大きいと考える．

4.3 Web アプリケーション実装例

実際に WebAPI を利用して Web アプリケーションを実装した例を図 5 に示す．

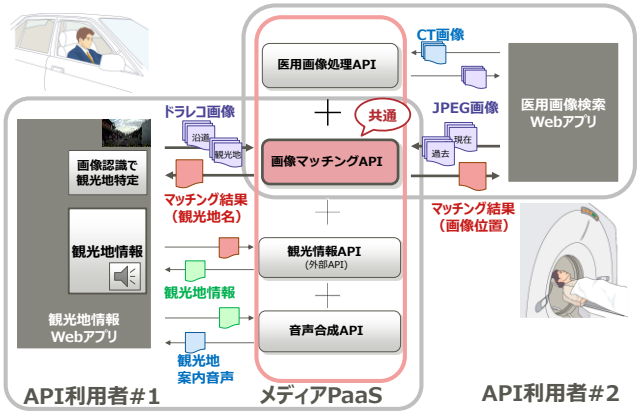

図 5　Web アプリケーション実装例

各 API を用い，観光地情報 Web アプリと医用画像検索 Web アプリを開発した．

外部 API も含め，複数の API を組み合わせることで目的のアプリケーション開発を実現できる．

また，画像マッチング API の例のように，機能が使いや

すい単位で部品化されているために，複数のアプリケーションから別々の用途で利用可能となっている．以下，各 Web アプリケーションの内容を紹介する．

4.3.1 観光地情報 Web アプリ

ドライブレコーダからの映像と観光地画像を画像マッチング API により比較することで観光地を検出し，その観光地に関して総務省の観光情報 API[*6]からデータを検索する．さらに検索結果の解説文字情報を，音声合成 API で音声変換し出力する．

図 6　観光地情報 Web アプリ

4.3.2 医用画像検索 Web アプリ

CT 画像などを使った医療分野における画像検査では，医師は病変部の経時変化を観察する場面がある[7]．そこで，過去の検査画像で見つかった病変部に対応する現在の病変部を自動で検索するアプリを開発した．

医用画像処理 API や画像マッチング API に対して，画像と病変部の位置情報(図 7，左部)を送ると，現在画像で対応する病変位置(図 7，右部)を，API を介して取得できる．このように API を組み合わせることで，医療向けの画像検査支援に使える画像検索機能を実現できる．

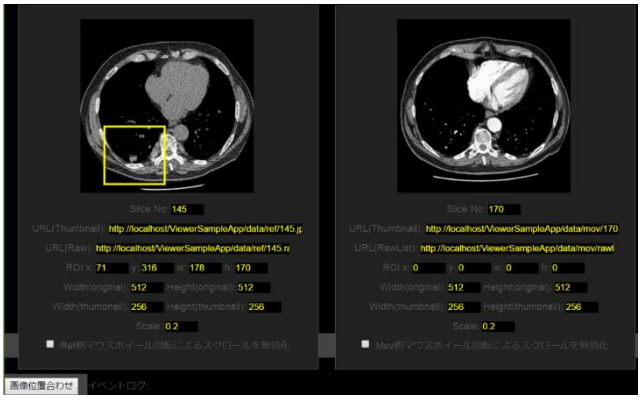

図 7　医用画像検索 Web アプリ

4.3.3 WebAPI アクセスの可視化

モニタリング用のコードをアプリケーションに埋め込むことで，各アプリケーションからアクセスされた API をグラフ形式でリアルタイムに表示できる．Web アプリケーションからの呼び出し例を図 8 に示す．このような呼び出しグラフがリアルタイムに更新され，アクセスパターンを可視化できる．

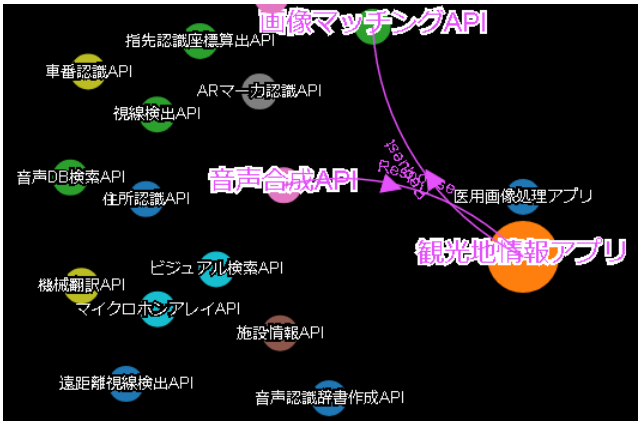

図 8　WebAPI アクセスの可視化

5. おわりに

5.1 まとめ

本稿では，映像や音声等のメディア処理の特性を考慮した WebAPI 開発・運用フレームワークを提案した．提案フレームワークではメディア処理技術の WebAPI 公開およびコンテナ化を容易に実現するためのテンプレートを提供した．また，メディア処理技術の開発・運用を，LC4RI・Ansible により管理する構成とし，運用効率化と配備先の柔軟性を実現した．開発・運用それぞれの面で効果測定を実施し，開発工数削減効果(コード量削減率 14.6%)，および運用コスト削減効果(サーバ全体の削減率 87.5%，API ごとの削減率 79.2%)を確認，提案フレームワークの有効性を確認した．

5.2 今後の課題

本稿で提案したフレームワークにおいて，テンプレートによるコード量削減効果はメディア処理により差があり，特にある程度複雑な処理を必要とするメディア処理に対しては相対的に効果が低い結果となった．これに対応するためには，メディアデータのコンテンツ内部，およびメディア処理自体の特性をより詳細に分析し，これに対応したテンプレートの提供が有効と考えられるため，検討を進めたい．

運用面では配備先の柔軟性をより活かせる構成として大容量メディアデータの解析・変換等を含むアプリケーションへ適用し，多様な環境へ配備可能であることの有効性

*6 総務省によるオープンデータ実証実験の一環として観光情報 API が公開された．なお本稿の表示例等で示すデータは 2017 年時点に取得したものである．
https://www.soumu.go.jp/menu_seisaku/ictseisaku/ictriyou/opendata/opendata03.html

を検証したいと考えている．

参考文献

[1] NII Cloud Operation Team “Literate computing for reproducible infrastructure.”, https://literate-computing.github.io/, (参照 2020-5-15)

[2] https://www.cdata.com/apiserver/, (参照 2020-5-15)

[3] 東野 正幸 “モノリシックサービスからマイクロサービスへの構造変更を容易化するためのウェブアプリケーションフレームワークの検討”, 情報処理学会研究報告, vol. 2018-DPS-174, no. 34, pp. 1-4, 2018.

[4] 中野 雄介, 池田 泰弘, 松尾 洋一, 渡辺敬志郎, 石橋 圭介, 西松 研 “機械学習を用いたネットワーク異常検知技術の WebAPI 化の研究”, 電子情報通信学会論文誌 B Vol. J102‐B No. 5 pp. 343‐355, 2019.

[5] 上野 優, 今井 祐二 “SSH Kernel: Jupyter Notebook でサーバの遠隔運用手順を記述・実行する Jupyter 拡張の開発”, 情報処理学会研究報告, Vol.2019-IOT-44 No.15 pp. 1-7, 2019.

[6] UENO Masaru, IMAI Yuji “SSH Kernel: A Jupyter Extension Specifically for Remote Infrastructure Administration” In Proceedings of IEEE/IFIP Network Operations and Management Symposium 2020.

[7] 石原 正樹, 小澤 亮夫, 松田 裕司, 杉村 昌彦, 武部 浩明, 遠藤 進, 馬場 孝之, 上原 祐介 “医師の画像診断業務を効率化する画像位置合わせ機能の開発”, デジタルプラクティス, 8(2), 情報処理学会, pp.144-151, 2017 年 4 月.