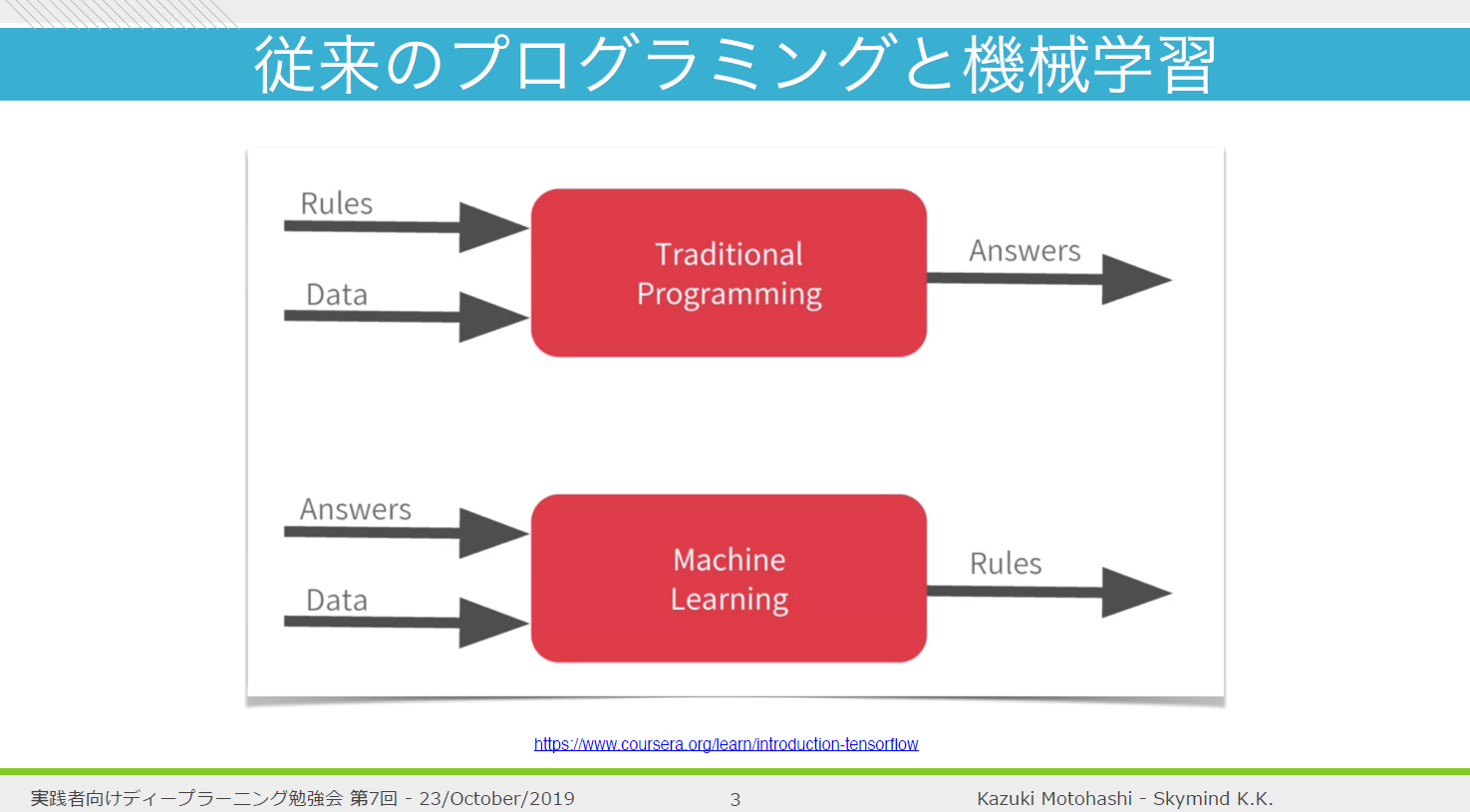
第４章　モデルの運用

機械学習にかぎらず実際に分析は一回行えば終わりというものではなく、日々新しい環境の変化や新しいデータが生成される中で、常にモデルがユーザにとって利用価値のある状態で維持するために運用をしていく必要があります。本章ではAIのコンテキストであまり語られることが多くないモデルの運用について説明をしていきます。

本章の本題に入る前に、本書はアプリケーションエンジニアのようにはじめて機械学習に触れる方もたくさんいらっしゃると思いますので、従来のプログラミングと機械学習について再度簡単におさらいをしておきます。

従来のプログラミングと機械学習ではパラダイムシフトがあります。従来のプログラミングは分析のルールとひたすら手でプログラミングしていって、それにデータを与えると答えが返ってくるというスキームになっています。それに対し、機械学習はデータと正解ラベルのような答えのペアを機械学習のアルゴリズムに流し込んであげて、ひたすら学習をしていくとルールを学んでいくというスキームになるというようにやり方の違いが存在します。つまり機械学習は手でルールを作る必要がなく、勝手にデータの傾向を学んでいくので、今までルールの記述が難しかった事象も自動化できる可能性が出てきています。このことにより機械学習やディープラーニングのブームが起こってきているのだと思っています。

特に東大の松尾先生などは、カンブリア爆発になぞらえ、生物が目を持つことにより自分で餌を探せるようになったように、コンピュータにAIの機能が実装されることで、あたかも目を持ったかのように自分でルールを学んでいくことが可能になるので、コンピュータやプログラミングの分野でより進化が進むと言われています。

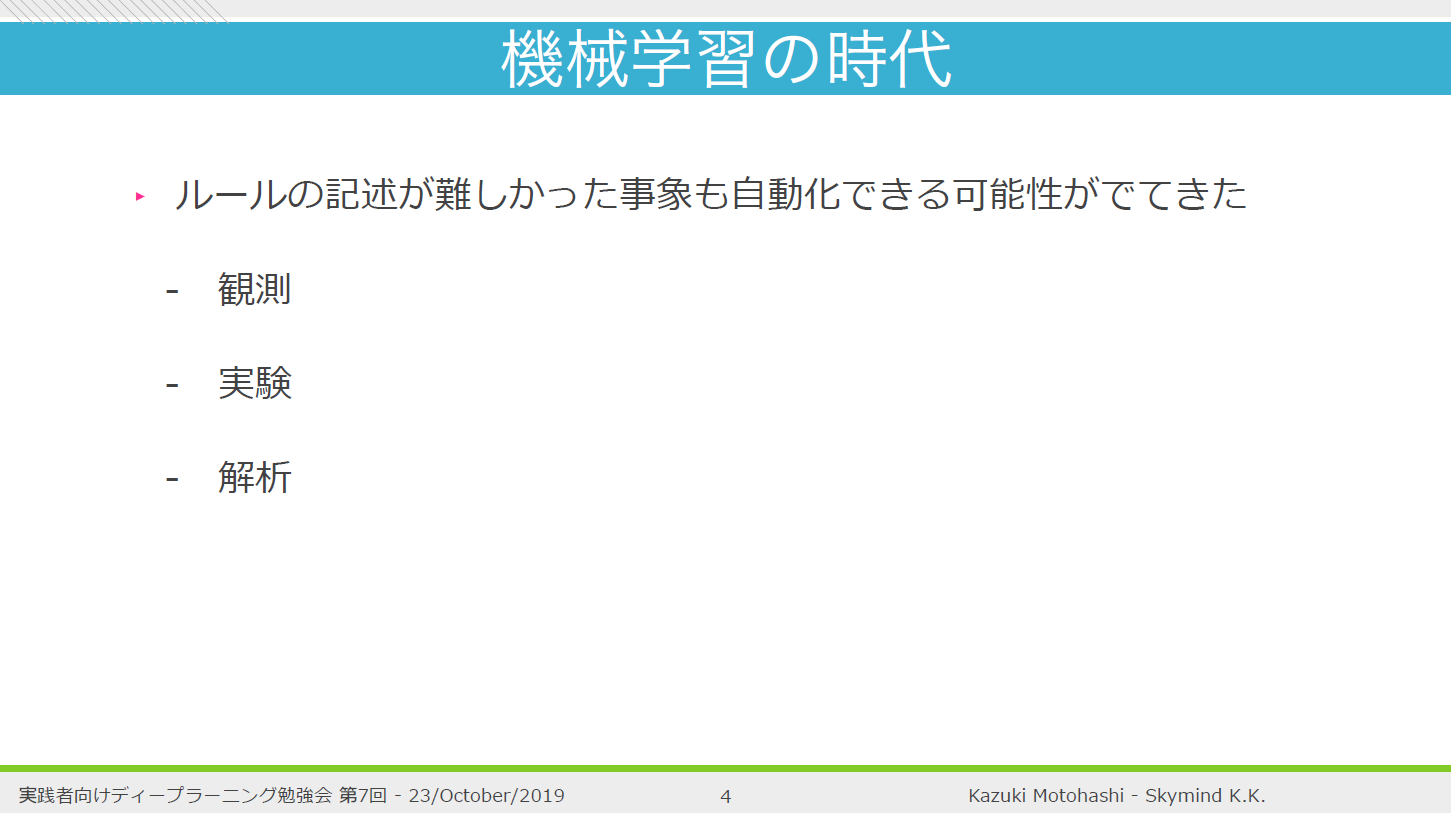


図： 既出なため載せるかも検討ください。

機械学習のライフサイクル

本節では、機械学習で必要となる一連の手順となる機械学習のライフサイクルについて詳しく見ていきます。

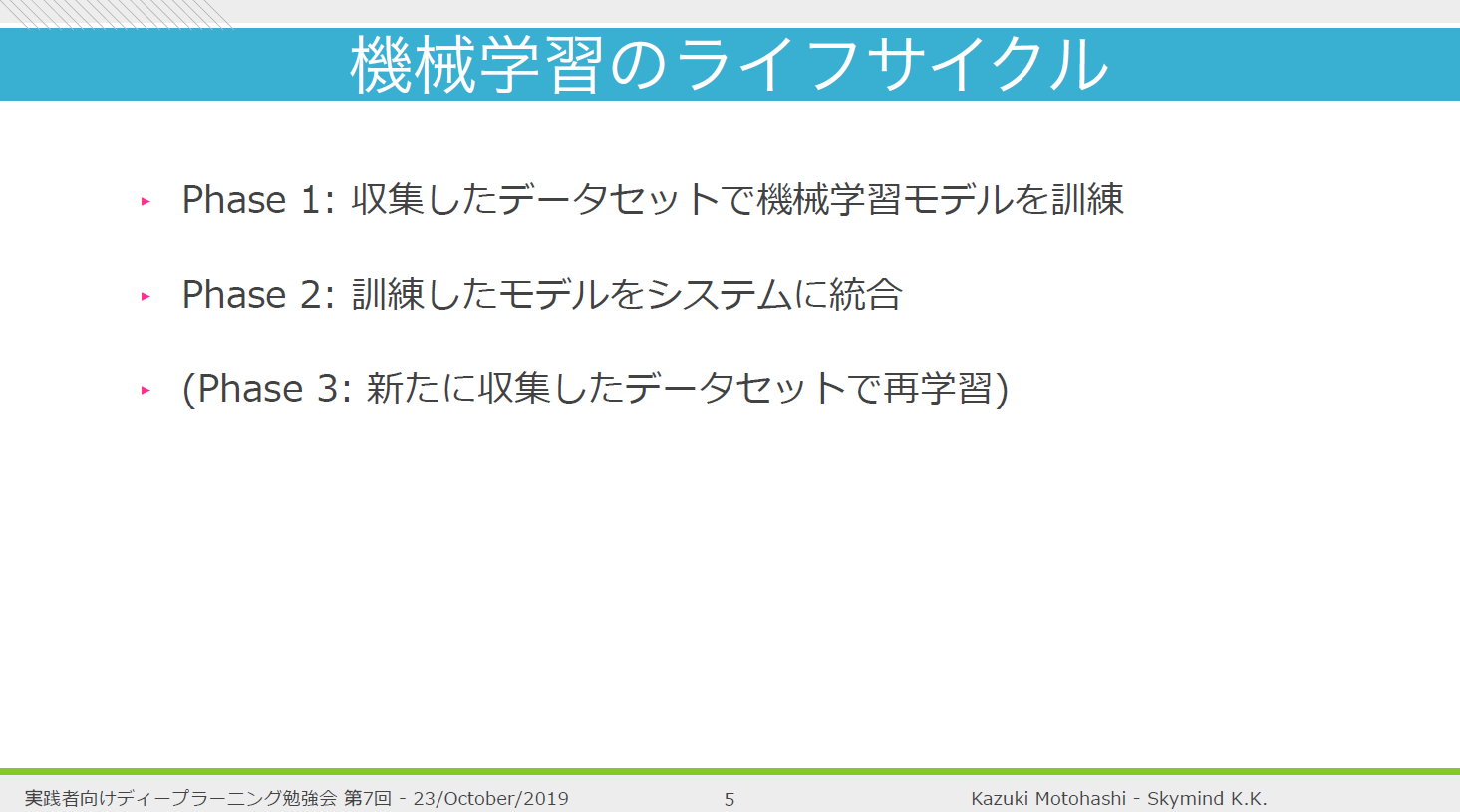
機械学習の開発の順序は最初に世界を観測します。つまりデータを集めて、そのデータに対して実験を行います。ここでいう実験とは、データをモデルに与えて学習を進めていくことです。次にその学習の結果の精度が良いのか悪いのかや、実用に値するのかそれとも何か変なバイアスがないのかなどを解析していきます。このフローは大学や大学院で研究をされた方であれば、そのときに行った研究のフローと同じようなことをやっているわけです。このことは、一般の方が研究者と同じようなことをやらなくてはならなくなったところまで来ているとも言えるわけです。



図： 機械学習の時代

つぎにもう少し具体的な話に入っていきます。

機械学習のライフサイクルには、下図のような３つのフェーズがあります。最初はデータの収集するところから始める場合も含まれますが、下図のPhase1で収集したデータセットに対して機械学習モデルを訓練していきます。その次にPhase2で訓練したモデルをシステムに統合するというところを考えなくてはなりません。もちろんそのあとには、Phase3として、運用をしていくうちに、どんどんモデルの精度が悪くなっていってしまうという性質があるため再学習を行います。どうして精度が悪くなってしまうかというと、データの傾向はたとえば時系列データなどでもそうですが、入力していくものが日々変化していくものなので、今まで学習したものと傾向もしくは性質が最初に学習した時点のものと異なることがあるためです。そこで何度も繰り返し学習をしてくというフィードバックのループを回すような仕組みが実際のシステムには必要になってくるわけです。実際の開発現場ではここまで考慮していないケースもあるかも知れませんが、少なくとも今後継続的に利用は必要となる場合には、Phase3のところまで考慮する必要があります。

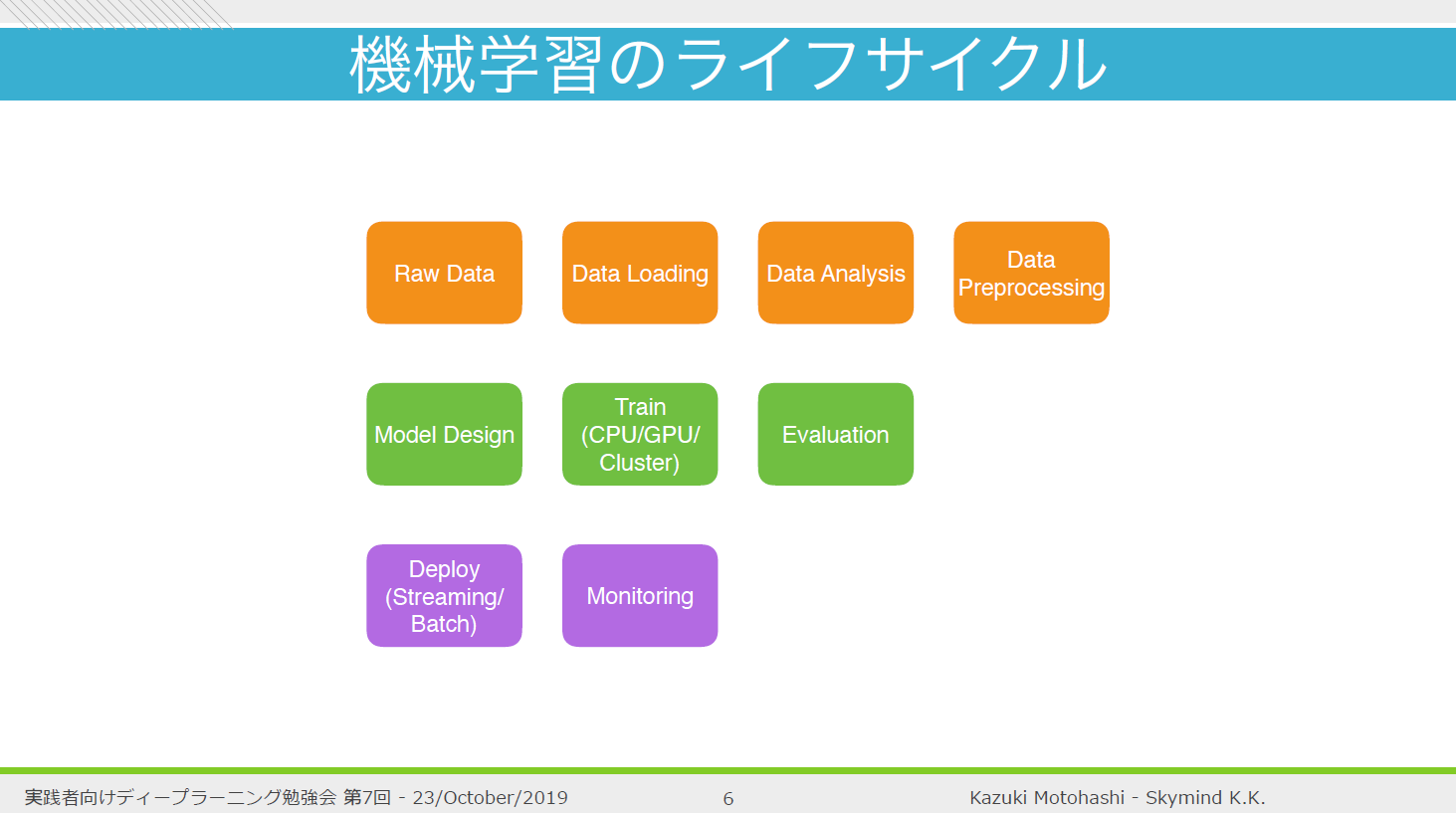


図：機械学習のライフサイクル概要

さらに機械学習のライフサイクルを細かく分けていくと、下図のようにたくさんのものがあります。下図の左がから右にフローが流れていくイメージですが、生データ（Raw Data）から始まりどんどん図の右下方向に進んで行くブロック図みたいなものだと理解してください。

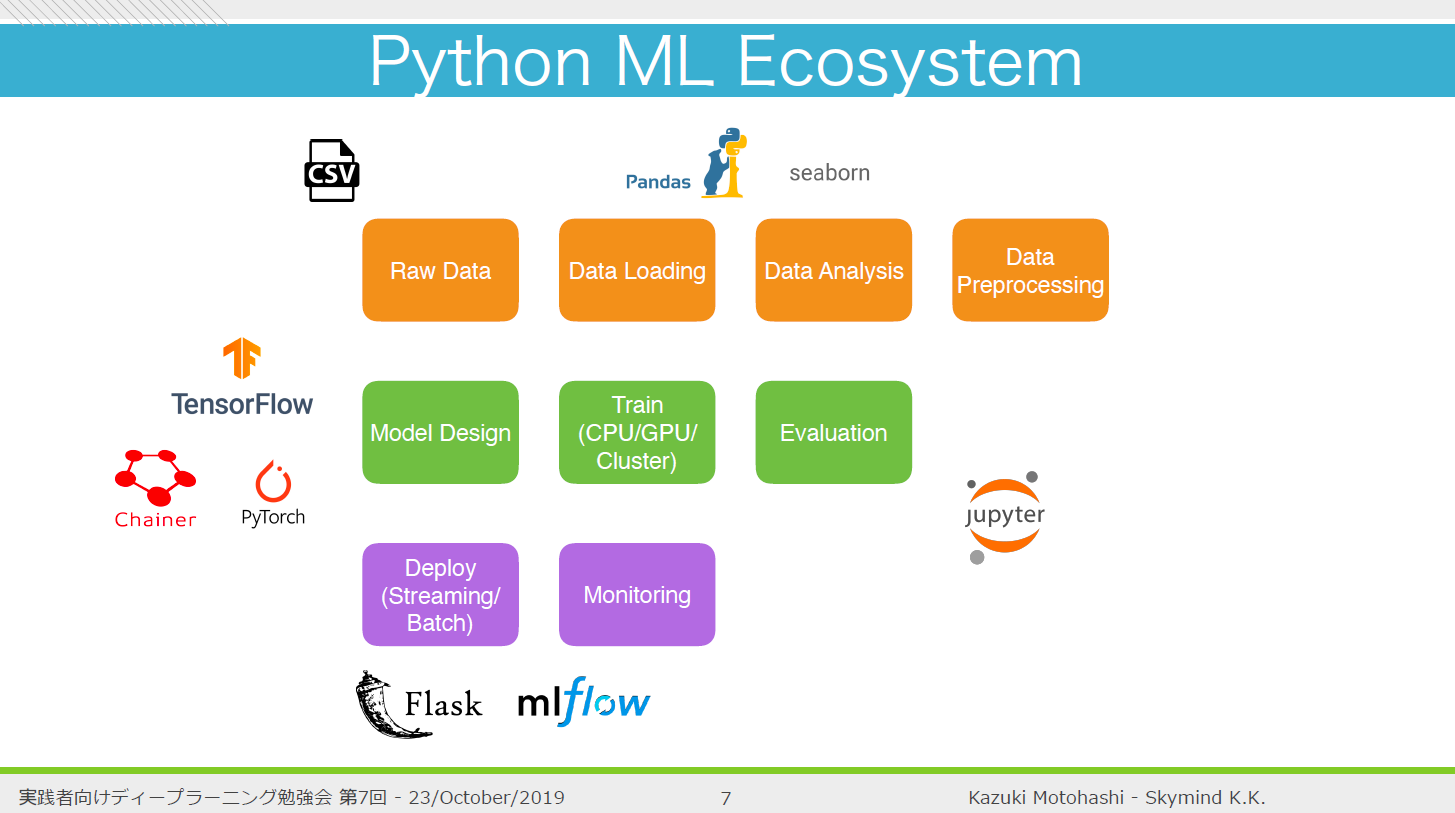
まず、生データがありそれを読み込んで、そのデータがどういった性質になっているかをなんとなく図示したり、平均や分散を見てみたりというようなことをひたすら実行するようなデータ分析（Data Analysis）を行います。機械学習モデルというのはたとえば画像分類を行う場合に、画像をそのまま入力するわけではなくて、機械学習モデルが扱いやすいように行列に変換したりデータを正規化したりといった何かしらの前処理が必要になります。

次に下図の1段下に移って、モデルの設計（Model Design）を行ってから、設計したモデルに対してデータを入力してトレーニング（Training）を行い、精度を評価して（Evaluation）、評価結果が望むべきものであったらさらに一段下に移って、モデルをデプロイ（Deploy）して、その性能やたとえばCPU/GPUの使用率などのトランザクションのデータをモニタリング（Monitoring）して運用を進めていきます。



図： 機械学習のライフサイクルの詳細

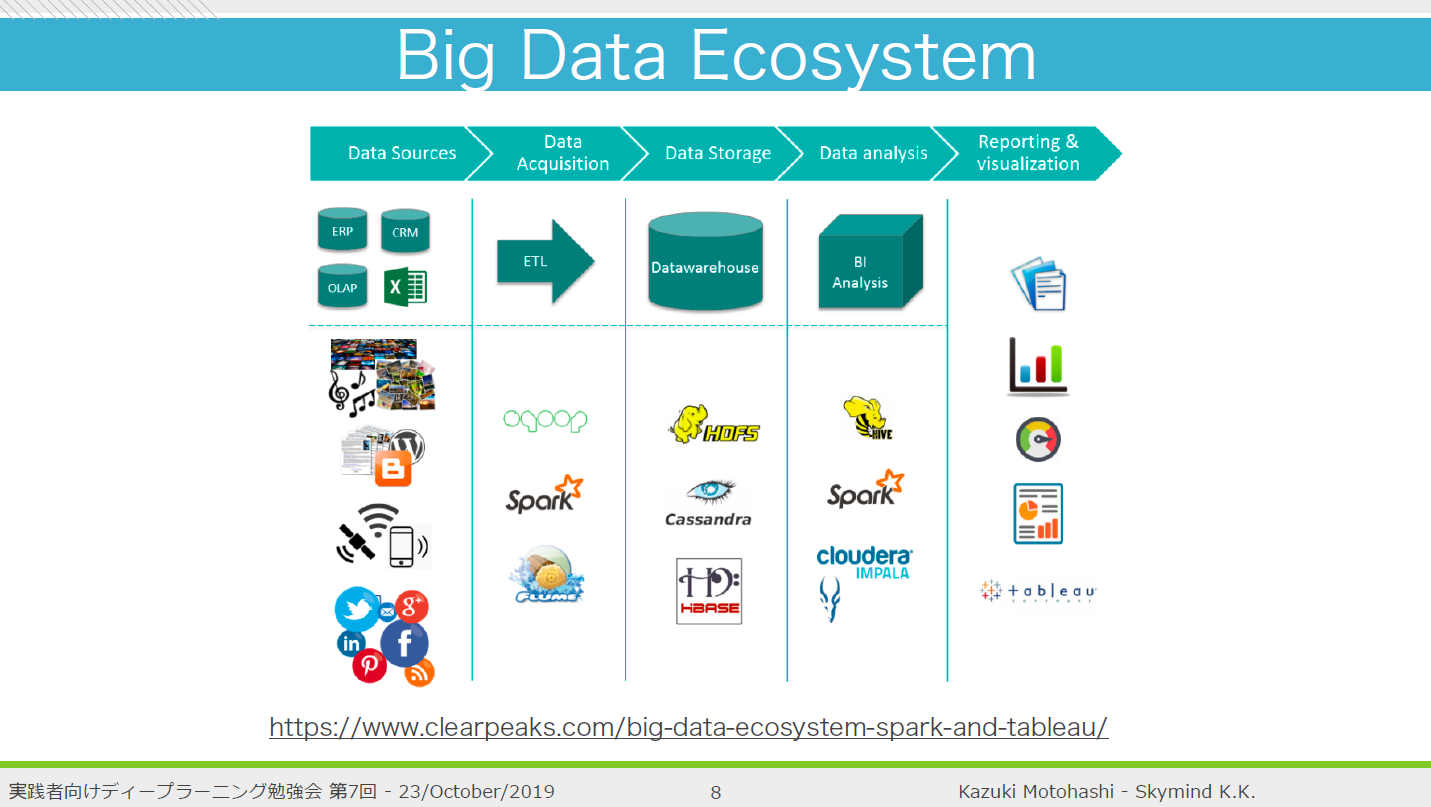
上記の図を読者の多くの方が利用されているPythonのエコシステムにどういうツールがあるのかを当てはめてみたものが下図になります。



図： 機械学習ライフサイクルで用いられるPythonエコシステムのツール例

データのロードに必要なPandasとかデータ分析に必要なseabornのようなものを用いて、機械学習の部分ではGoogleのTensorFlowや、Preferred NetworksのChainerやFacebookのPytorchを使っているわけです。実験に適した環境としてはJupyter Notebookに代表されるインタラクティブな開発も可能なノートブックの環境などでモデルを開発していきます。次に出来上がったモデルは何らかの形でシステムからアクセスできるようにデプロイするわけです。その場合によく利用されるのがFlaskのWebアプリケーションのフレームワークです。Flaskが良く用いられている理由は、その処理が軽いのと扱いやすさにあると思います。ただしもう少し凝ったことを行いたい場合には、mlflowのようなモデル開発の全工程を管理するようなツールを使ったりします。このような仕組みが良く使われていると判断している理由は、みなさんKaggleなどのデータ分析のコンペティションでさまざまな手法が公開されるわけですが、そういったところでこの仕組みが良く用いられているからです。ただしKaggleなどはみなさんが取り組みやすいようにCSVやGoogleが主催しているためBigqueryなどの扱いやすいデータ形式を使っています。

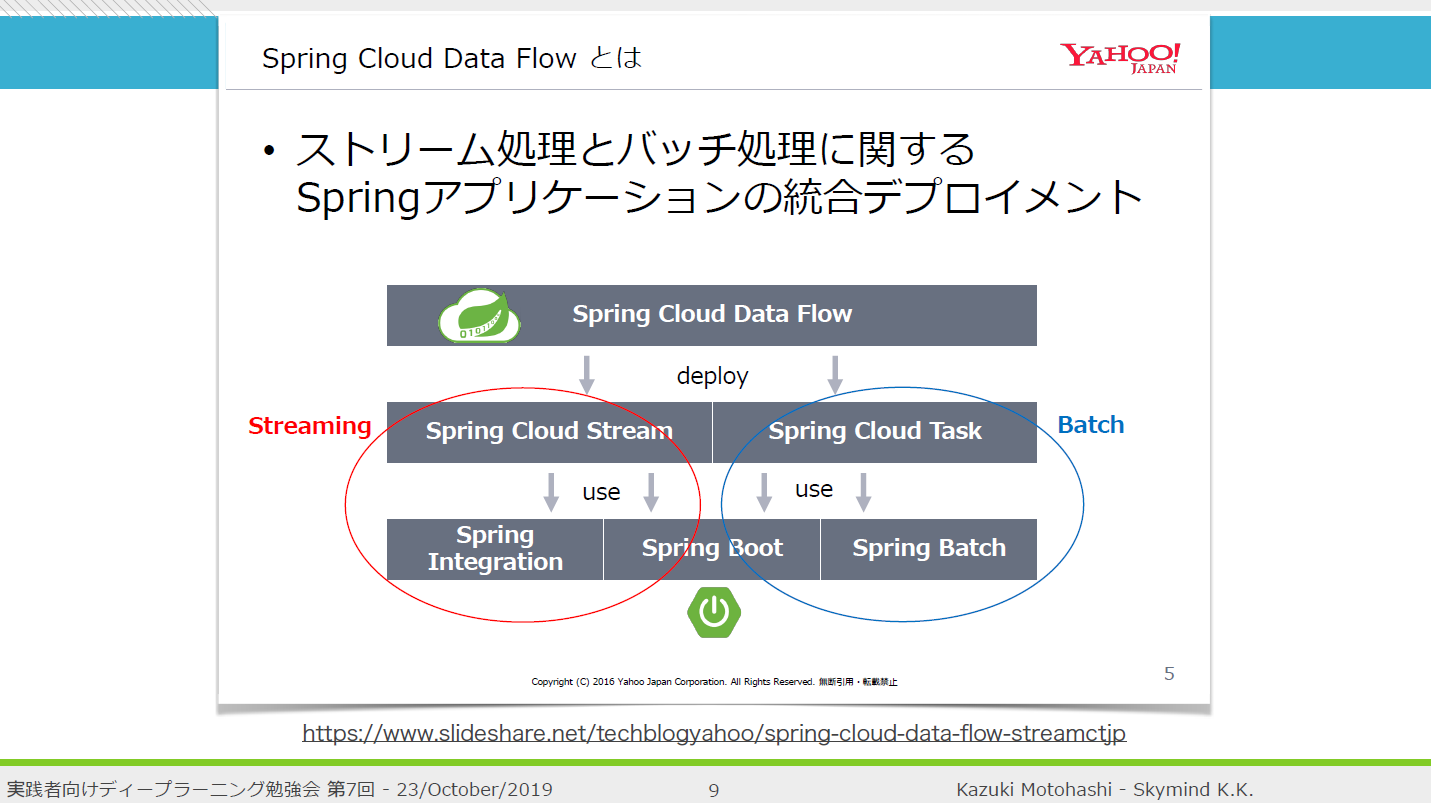
しかしながら大きな企業などで使われているようなデータの保存形式はもちろん小さいチーム等ではcsvが使われたりしますが会社全体では一般的なわけではなく、たとえば顧客管理のさまざまなERP系のツールや、SNSなどいろんなところからデータが溜まっていくわけです。そのデータを何かしらのビッグデータに適したようなデータベースに貯めていきます。ただ単にすべてを保存していくだけですと非常に無駄が多いので、Sparkなどを使ってデータ抽出・変換・加工にあたるETL（Extract-Transform-Load）処理を用いてある程度軽量化したうえで、データウェアハウスであるHadoopやCassandraやHBaseのようなところに保存していきます。ひとたびデータが蓄積されると、SQL文などを記述して可視化するためのBIツールに入力するとった下図のような構成を組んでいることと思います。もちろんこれらの構成はオンプレミスだけではなくて、AWSのようなクラウドサービスではこれと同じようなことを手軽に実現しさらにインフラ管理までしてくれるため、それらを使ってらっしゃる方もいると思います。前のPythonのエコシステムの例とビッグデータのエコシステムの図を見ていただければわかると思いますが、２つの図にはかなりのギャップがあります。つまりKaggleのようなコンペティションと実際の企業での利用ではギャップが存在します。しかも大企業やエンタープライズ系の会社ではJavaを使う機会が非常に多いと思います。そこでこれから企業ユーザが使用する場面を想定しJavaの利用について話を進めていきます。



図： ビッグデータのエコシステム概要

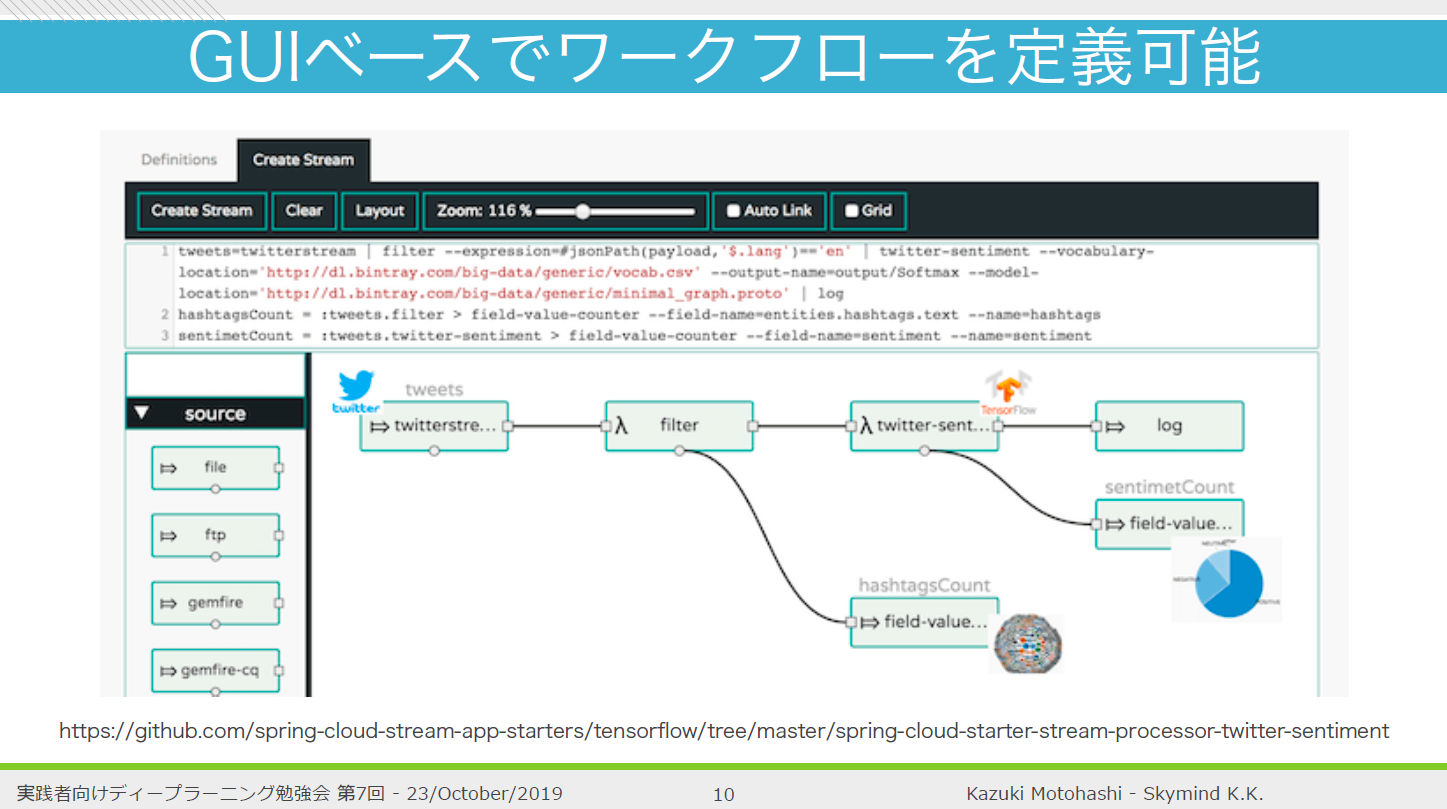
Javaでの開発環境とは

Javaで広く使われているWebアプリケーション開発用のSpringフレームワークがありますが、その中の１つのアプリケーションにSpring Cloud Data Flowというものが存在します。これが何をしてくれるものなのかというと、以下の図はYahooのスライドから引用したものになりますが、Spring Cloud Data Flowは、リアルタイムでデータが入ってくるストリーム処理や溜まったデータをバッチ処理するためのアプリケーションです。このアプリケーションは、下図のようにストリーム処理用のSpring Cloud Streamとバッチ処理用のSpring Batch Taskの2つから構成されていて、さらにそのバックエンドとしてSpring Boostというもので構成されています。



図： Spring Cloud Data Flowの概要

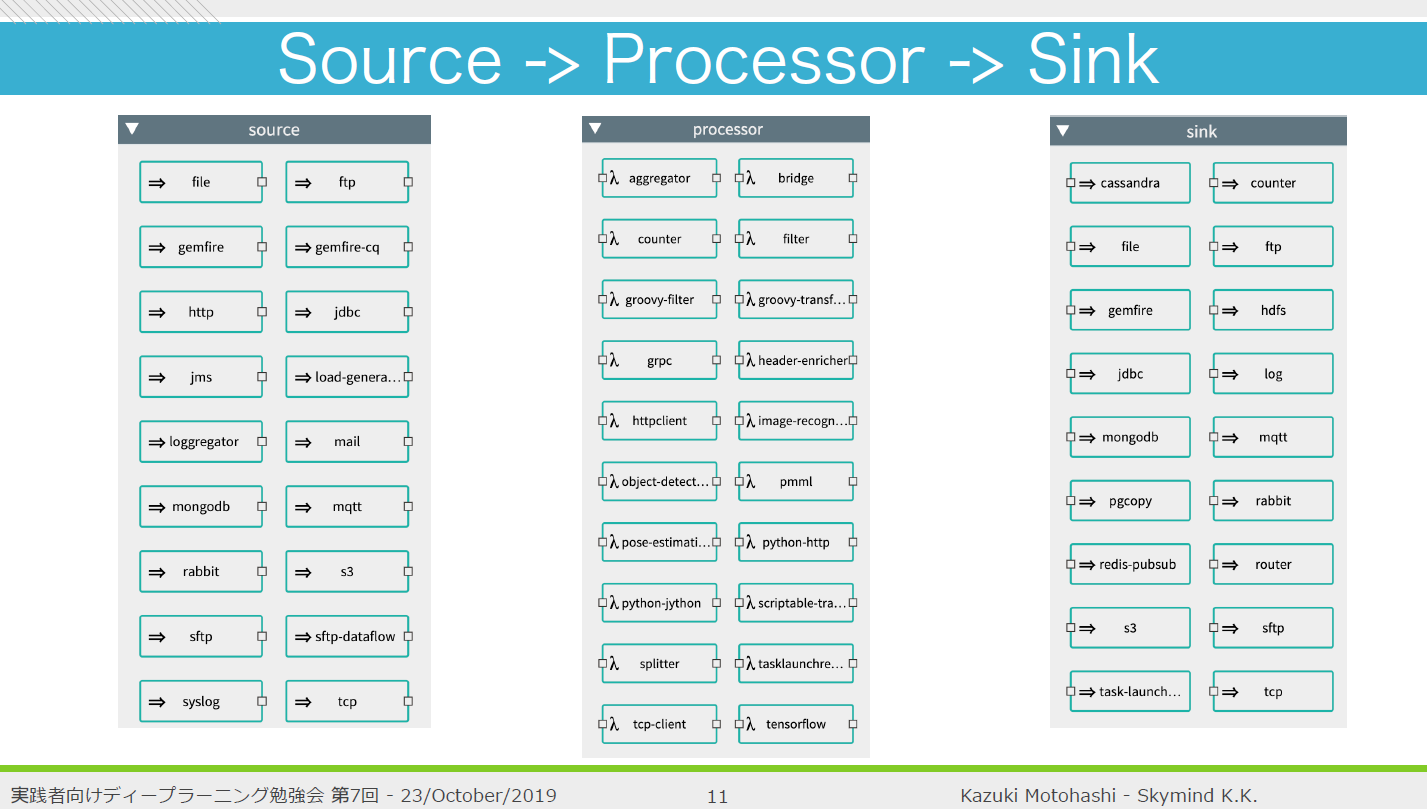
上記の説明ですと実際のイメージがしづらいと思いますので、下図に示されるGUIベースをもとに説明します。こちらの例は、下図の下のブロック図を見ていただけるとイメージがしやすいと思いますが、ツイッターのタイムランからツイートをどんどん引っ張ってきて、そのデータに対して何かしらのフィルタリング処理をかけて、最終的にログとして出力をしているというようなシンプルなものになっています。実際にはこの図の中では、フィルタリングとしてハッシュタグだけを取ってきて、ハッシュタグがいっぱいある中で一番盛り上がっているハッシュタグがどれなのかをGrafanaという可視化ツールでグラフに表示していたり、ツイートそのものをTensorFlowのモデルに入力して、感情分析をしたりしています。ちなみに感情分析が何かというと、たとえば今日はラーメンを食べておいしかったというような場合には感情としてはポジティブというようなものを分析するというものです。最終的にはその感情分析結果をここではログに入れたり、センチメントがポジティブなのかニュートラルなのかネガティブなのかを円グラフで表してみたりということをやっています。この例は、下のURLにオープンソースで公開されているので、ご興味がある方は見てみてください。こういうフローを下図の上の部分のプログラム記述例のドメイン固有言語（Domain Specific Language）で記述して、twitterstream | filter + 引数を与えて | twittersentimentのようなものをどんどん並べていくことで、こういった複雑なフローが完成するというような優れた機能を持っています。ちなみに | はパイプと呼ばれ、コマンドをパイプでつなぐことにより前のコマンドの出力を次のコマンドの入力として引き渡すような機能で、Linuxなどでは一般的に用いられているものです。



図： GUIベースのワークフローの例

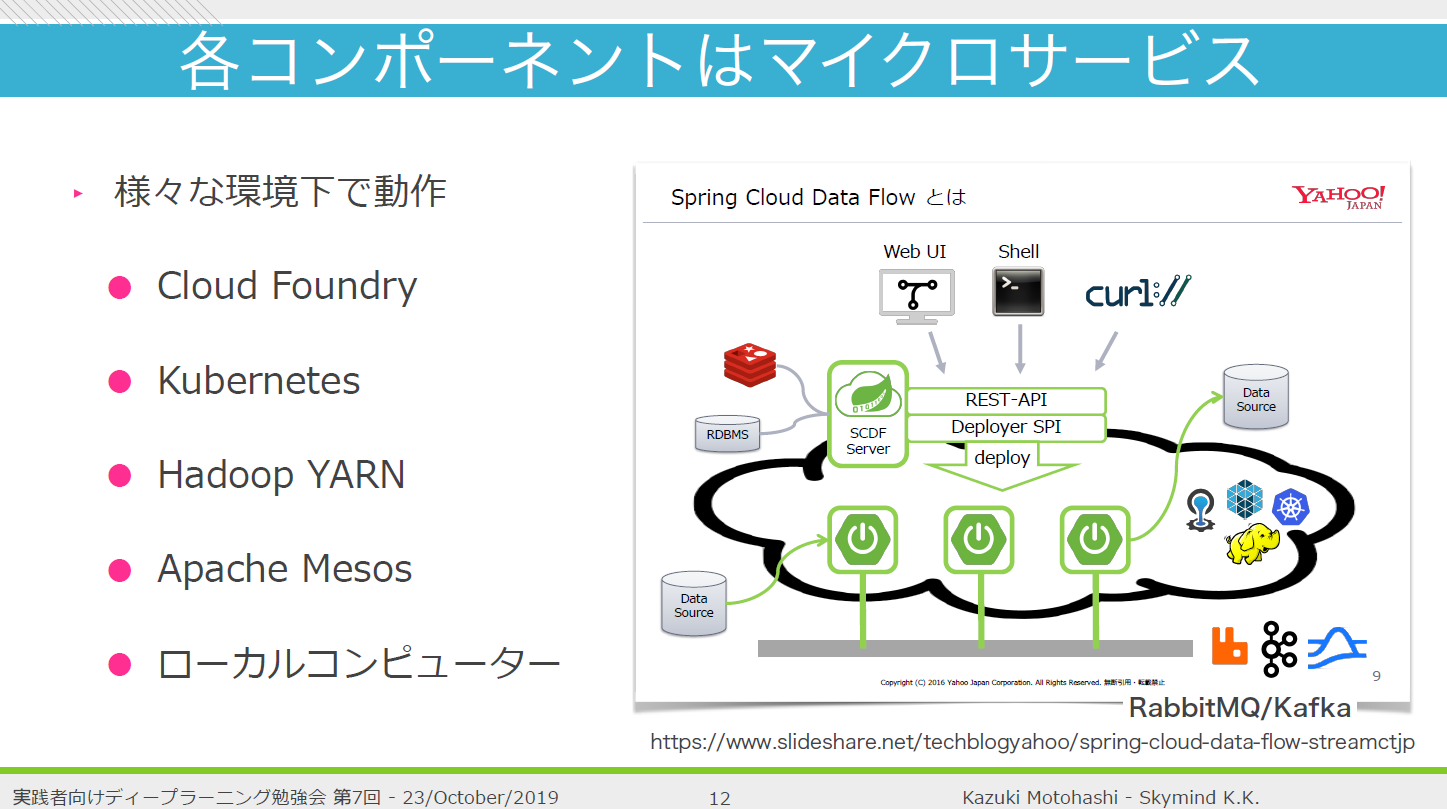
Spring Cloud Data Flowにはすでに下図のようないろんなブロックがデフォルトで入っていて非常に便利です。以下の例はあるディレクトリを監視してそのディレクトリ内に何かのファイルが入ってきたらそれを次のストリームに流すというようなファイルストリームのようなことが簡単にできます。

図の左のSourceにあるボックスには、ファイル転送系のftpなどのほかに、データベース系のjdbcやMongDBのようなものや、IoTで良く利用されるMQTTや、メールや、クラウドストレージとして良く使われているS3のようなものがあり、それらのコンポーネントの中から使用するものをSourceとして配置します。次にProcessorのボックスでは、counterやfilterや別のサーバにリクエストを投げるためのfttpclientや、pmmlやTensorFlowなどではProcessorのモデルを指定してあげれば入力から各Processorのモデルに入れて出力を次のフローに流していくというようなことができたりするような機能があります。図の右のSinkは出力先を意味する機能で、Cassandraのようなビッグデータ基盤とかfileとかでシンプルにログに出すとか、Sourceおと同じになりますがMongoDBのようなものも実装されています。このような機能をそれぞれつなぎ合わせていけばアプリケーションが完成するという仕組みになっています。もちろんご自身で各ブロックのコンポーネントとして新たに機能をSpring Boostのフレームワークの規則に則って開発していけば、わりと簡単にコンポーネントを作ることができます。



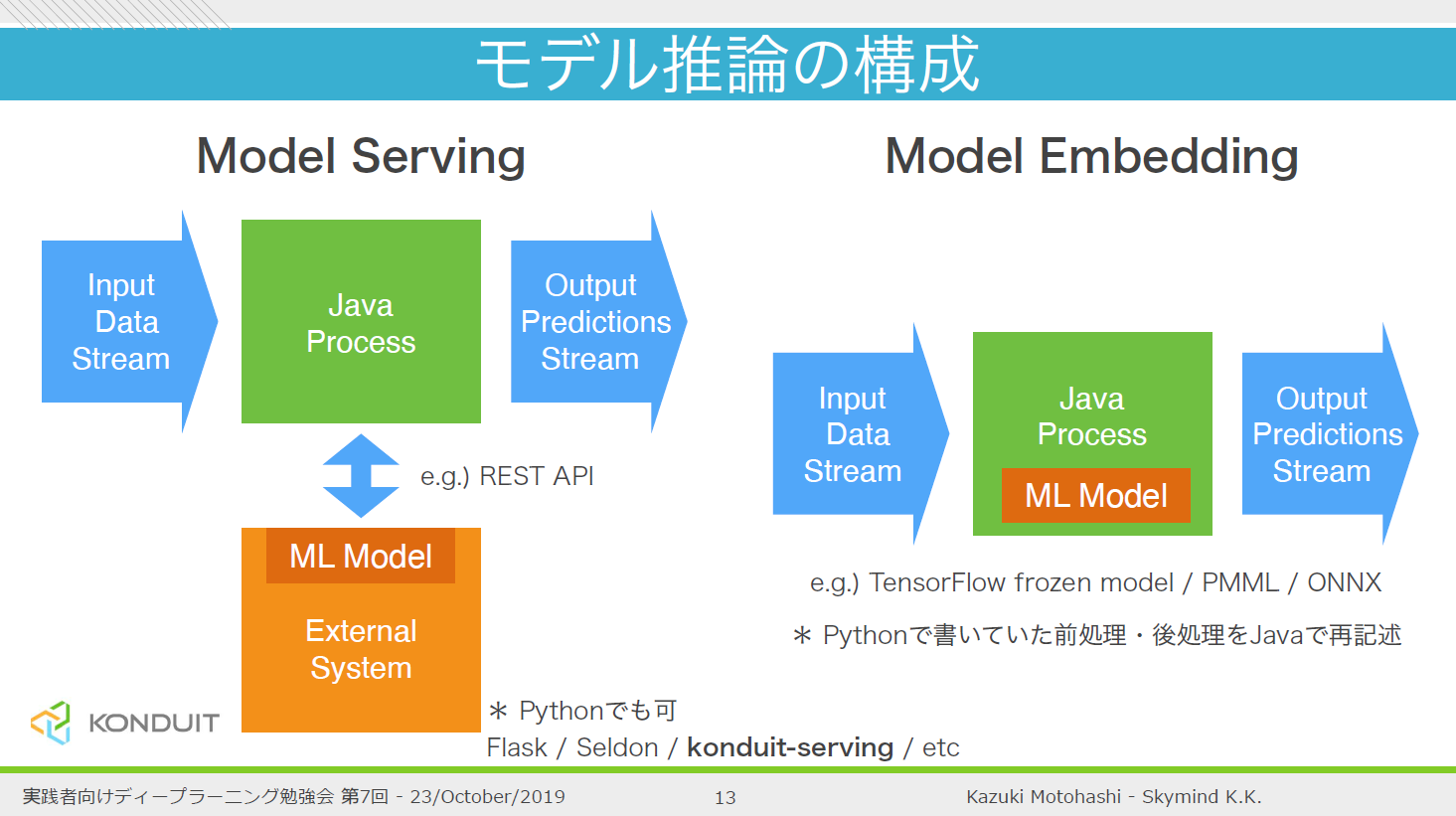
図： Spring Cloud Data Flowのコンポーネント概要

これらの各コンポーネントはマイクロサービスとして独立に動作します。下図の右の図では、一番下の層にRabbitMQやKafkaのメッセージングのツールがあって、それぞれのコンポーネントが各メッセージングと接続されています。そのメッセージングを使って各コンポーネント同士が通信を行うことによってフローが形成され、さきほどの見てきたGUIベースのWeb UIのようなツールから作業することもあれば、コマンドラインベースのシェルから開発することもできます。さらにデプロイした環境に関しては、Curlやhttpのクライアント経由でアクセスすることができたり、外部のデータベースと繋げたりといったことができます。このソフトウェアにCloud Data Flowという名前がついている通り、Croud FoundryやKurbernetesといったクラウドネーティブなツールであったり、Hadoop YARNやApache Mesosといったオーケストレータのもとで動作するもので、ローカルマシンから開発を始めても大丈夫ですし、多数のサーバに対してスケールアウトすることも可能な仕組みになっています。



図： マイクロサービスに関する説明

次に上記のようなものを使ってどうやって機械学習やディープラーニングのモデルを統合していく方法として、大まかに分けて下図のような2つのものがあります。1つ目はモデルサービング（Model Serving）と呼ばれるもので、もう一つはモデルエンベディング（Model Embedding）と呼ばれるものです。実際の内容としてはさほど複雑なものではありません。1つ目のモデルサービングは、下図の左のように、データがあってここの例ではJavaのプロセスにそのデータを入れています。そこのプロセスが別の場所にあるサーバのREST APIなどに対してリクエストとデータを投げます。Java Processの外側にあるAPIサーバ上で推論を実行し、その推論結果をJava Processに返して後ろのストリームに流していくものです。それに対し、2つ目のモデルエンベディングは、下図の右のように、たとえばTensorFlow for JavaやDeeplearning4jといったようなものを使って、ローカルにあるJavaのコード自体にモデルを組み込んでいていくというものです。具体的にはTensorFlowのフローズンモデル（開発済みのモデル）やPMMLやONNXなどを使って行うというものです。ただしモデルエンベディングの方には若干の欠点があります。それが何かというと、機械学習モデルというのはPythonの方がエコシステムが充実しているためPythonで開発すべきだと思いますが、実際にはモデルを配布して終わりというものではなく、それをJavaのコードで書き直すという処理が必要となります。そのためモデルに対してデータを入力するための前処理であったり、たとえば画像分類の例で言うと“これは犬です”というような場合には“犬”と出力してくれる方が数字の羅列で記述されるよりわかりやすいので、その後ろの行列とかテンソルを人間が読める形式に変換したりしてあげるような後処理をJavaで再記述し直す必要があるという欠点があることです。もちろんモデルエンベディングでこのような処理を実現している企業などもいらっしゃいますが、広く用いられているのは流行という点を踏まえてもモデルサービングの方です。モデルサービングの方にはさまざまなフレームワークが存在していて、さきほど紹介したFlaskであったり、Kubernetesで機械学習を実現するためのパッケージであるKubeflowで使われているSeldonであったり、今回紹介するkonduit-servingというようなものを使ってサービス化をしていきます。



図： モデル推論の実装方法

konduit-servingとは

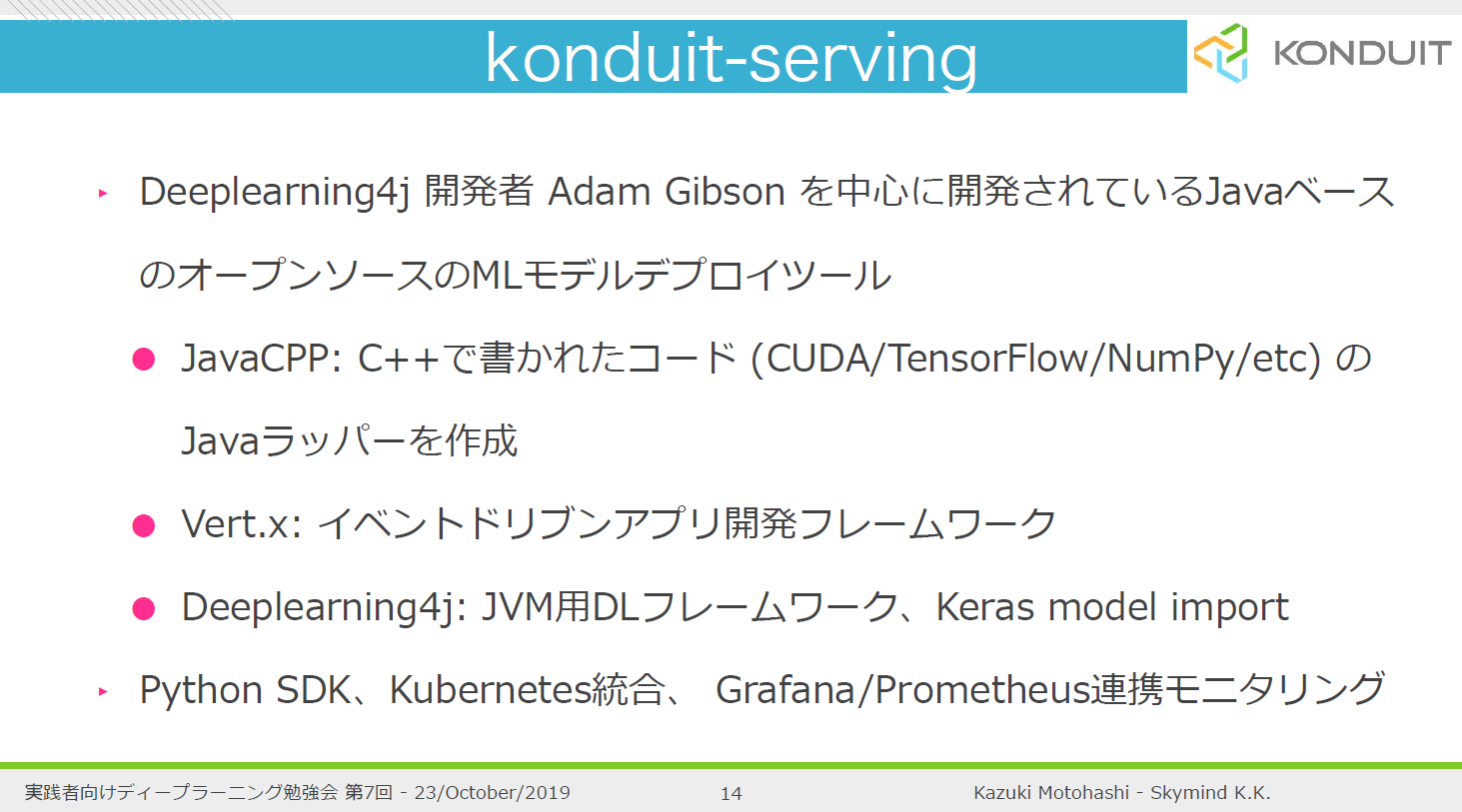
konduit-servingとは、Deeplearning4jというJavaやJVMの環境で利用可能なディープラーニングのフレームワークの開発者が開発したJavaベースのオープンソースのMLモデルのデプロイツールです。こちらのコアとして用いられているライブラリに下図に示されるJavaCPP, Vert.x、Deeplearning4jがあります。

JavaCPPはC++で書かれたコードをJavaで使用可能にしてくれるものです。通常、GPUとのコミュニケーションではCとかC++のネーティブな言語を用いる必要がありますが、このライブラリがあることで、Javaの仮想マシンをかませた言語でも使えるものになります。JavaCPPはまた、TensorFlowとかNympyなどPythonの世界で使われているがバックエンドの中でC++で書かれているようなライブラリをJavaにそのまま持ってくることができるができます。

次に、Vert.xはイベンドドリブンのAPIサーバを作るためのアプリケーション開発フレームワークで、かつ高速でスケールすることが可能な通信を実現可能なものです。

最後のDeeplearning4jはもともとkonduit-servingの開発者がDeeplearning4jの作者でありさらに、ディープラーニングのフレームワークでディープラーニング関連の演算であったり、Kerasのモデルをインポートしたりとかの機能が実装されているため、このフレームワークも使用されています。

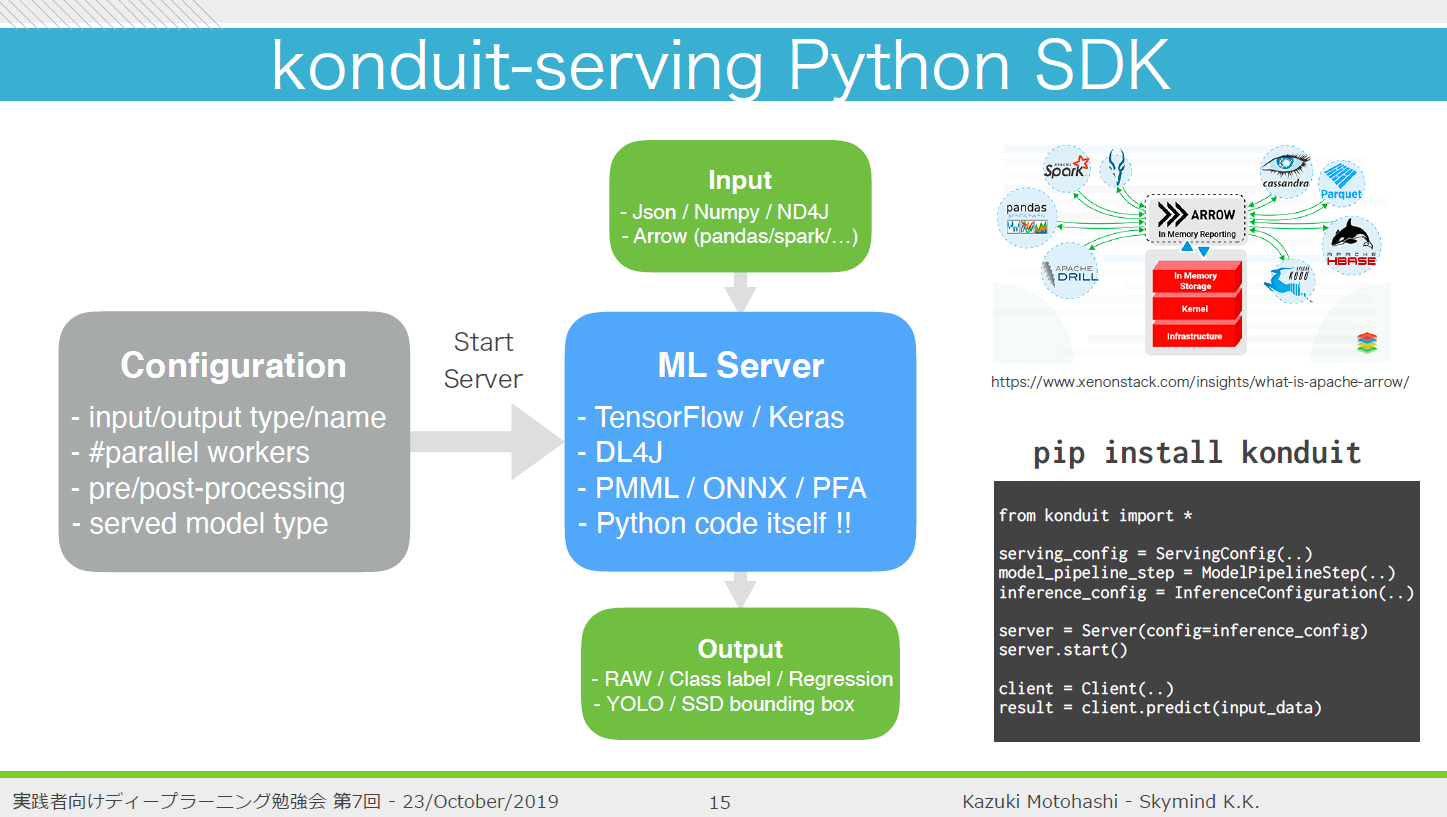
konduit-servingには、さらにPython SDKが実装されているので、Pythonユーザの方でも使い易いですし、Kubernetesとの統合もでき、Grafanaという可視化ツールやPrometheusというモニタリング関連のソフトウェアとの連携もサポートしています。



図： konduit-servingとは

これからkonduit-servingで実際に開発していく流れについて説明します。まずは下図の一番左のConfigurationのところでは、モデルをサーブ（提供）するために必要な構成の定義を行います。具体的には、どういう形式のものを入力として用いるのかとかどういったものを出力するのか、それらの型はどのようなものか、その変数は何なのかその名前は何かなどを指定して、並列に立ち上げるために必要なparallel workersの数を指定したり、前処理および後処理でどういうものを使うのか、TensorFlowのモデルなのかKerasのモデルなのかなどのサーブすべきモデルのタイプを指定したりします。このようなものをコンフィギュレーションとして記述します。

下図の右下のものは、実際のコードのうちだいぶん省略したものになりますが、まずModelPipelineStepやInferenceConfigurationなどのコンフィギュレーションを書いて、それをServerというコンストラクタに食わせます。あとはserver.startを指定することでサーバが立ち上がります。これに対してクライアントからリクエストを投げてあげれば結果が返ってくるというような非常に簡単な仕組みになっています。



図： conduit-servingのPython SDKの概要

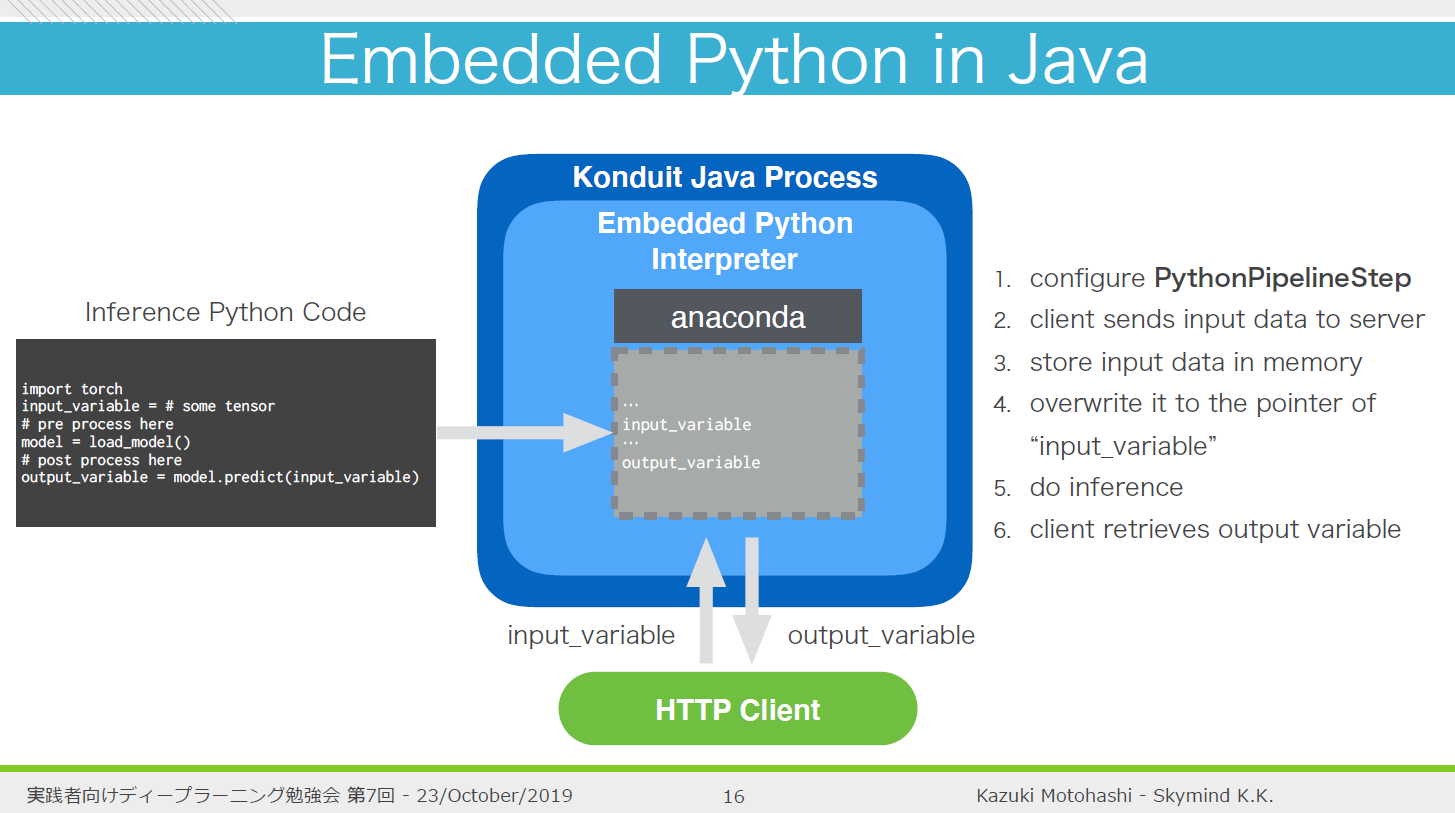
インプットの形式としてはWeb系のhttpのものなのでJsonをサポートしていますし、またNumpyをそのまま使うことができるので、Numpyの配列をそのまま投げることができます。さらには、ND4JというJava版のNumpyも使うことができますし、Apache Arrowというインメモリの列指向のデータフォーマットもサポートしていて、これらはPandasとかSpark関連のDataFrameをインメモリで非常に高速化可能にするために使用することができます。それらのインプットを今立ち上げた図の真ん中の四角に示されるML Serverに投げてアウトプットが行列であったり後処理したクラスラベルであったり、回帰分析の結果の場合には単なる数値データになりますが、そのようなものを返すソケットがkonduit-servingには用意されています。たとえば、何か物体認識をさせたい場合には、出力結果としてどこに何が写っているのかを境界ボックスの情報を返して欲しいわけですが、ある場所から別の場所までのピクセルに人が写っていますといった情報を返してくれるようなアウトプットのソケットも用意されています。

ML Serverがサーブできるモデルの種類としては、TensorFlowとかKerasはもちろん、Deeplerning4jがあります。さらには、PMMLの形式がサポートされているので、Scikit-learnなどで作ったモデルも使えますし、ONNX経由でPytorchやChainerで記述されたモデルを使うことができます。

またKonduit-servingの非常に面白い機能の１つが、PythonコードそのものをAPI化することがでるというものです。そこで以下でこの機能について説明をします。

konduit-servingでのネーティブなPythonコードの利用方法

konduit-servingには下図の真ん中に示されているように、konduit Java Process内に組み込まれた（Embedされた）Pythonインタープリタというのがあります。これは簡単にいうとJavaの上で動くPythonです。ここにPythonコードを与えてあげると、データサイエンティストが作ってくれたPythonで記述されている前処理や後処理の工程をすべて、そのまま運用に利用することが可能になります。



図： konduit-servingでのPythonコードそのものの利用方法の概要

実際の使い方としては、最初にPythonPipelineStepというコンフィギュレーションを設定してあげて、次にサーブしたPythonコードに対してクライアントから入力データをinput\_variableに対して投げてあげます。そうするとJavaの中のメモリにそのデータが格納されます。格納されたデータをinput\_variableのポインタに対して入力してあげることで、データ自体を別にコピーすることなく、つまりゼロコピーで推論まで進むことができます。さらに推論結果をPythonコードの中のoutput\_variableに入れれば、それを引っ張ってくることができるというような仕組みで、データやパラメータなどの入出力のやり取りが簡単にかつ高速に行えるようになっています。

以上が本章の主な説明になりますが、以下で今まで学んだ一連の機能を使った実際のデモについて説明します。

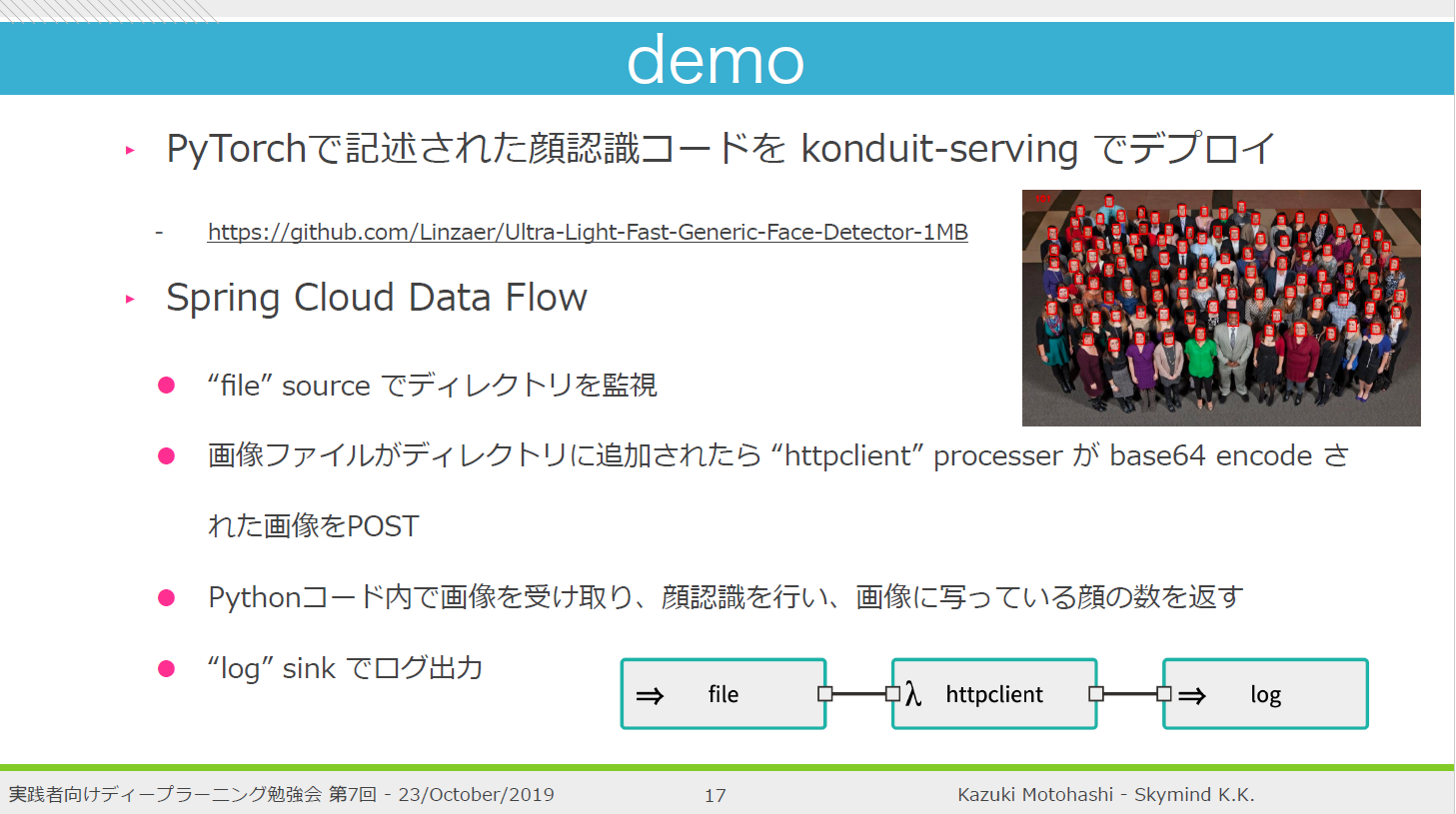
一連の機能のデモ

最初に今回実装するデモの概要を説明します。

今回のデモで用いるモデルは、PytorchというFacebookのディープラーニングのフレームワークで記述された顔認識のコードで、それをkonduit-servingでデプロイして、Spring Cloud Data Flowを使ってアプリケーション化するというような一連の流れを見ていきます。

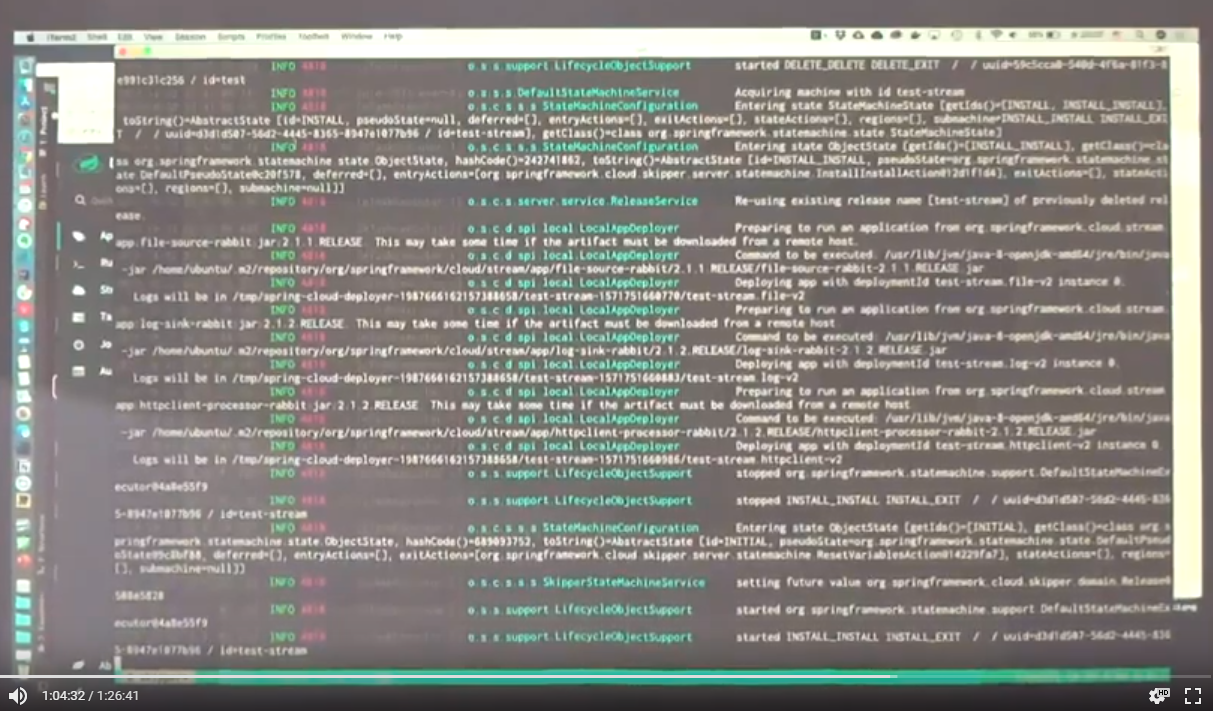
使用するモデルは、数日前にツイッターでバズっていた中国の型が作ったGithubのコードであるのですけれども、モデルのサイズが1MBという非常に小さいもので非常に高速に動くと言っていたものです。実際、下図の右のたくさんの人が並んでいるような画像でもそれぞれの人の顔を認識して赤い四角が全員分ついていてこのようにそれぞれの人が小さく写っていてもすごい精度で認識をできるようなモデルです。このデモでは下図に示されるこのGithubのモデルをデプロイします。

Spring Cloud Data Flowの中でやっていることは非常に簡単で、別にデータベースなどを使うこともなく、下図のフロー図のようにfileっていうブロックで、あるディレクトリを監視します。具体的にはimageというディレクトリを作成していてそのディレクトリを監視しておいて、そこに画像ファイルを追加したら、次のステップに進んでhttpclientを使ってhttpのクライアントがbase64ベースの文字列にエンコードされた画像をポストします。次にデプロイされたPythonコード内で画像を受け取って顔認識を行って、画像に移っている人の顔の数を返し、最後にログに出力するといった非常に簡単なものです。



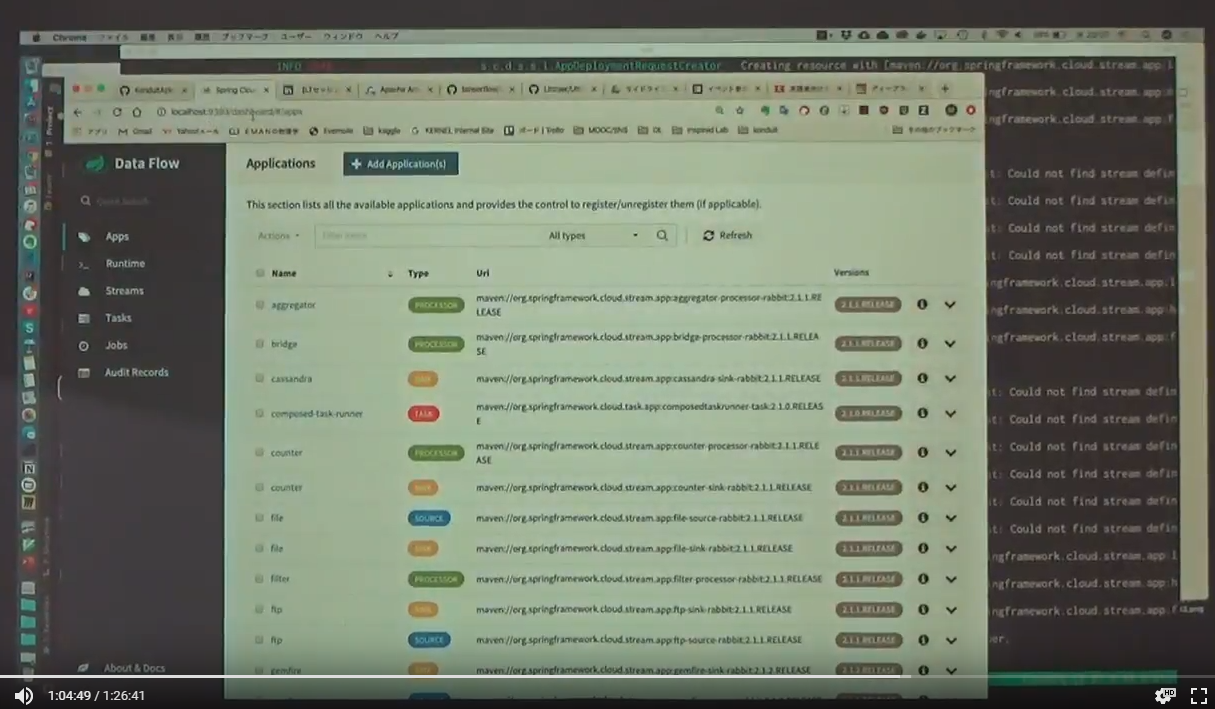
図： 今回のデモの概要

Spring Cloud Data Flowに関しては、普通のJavaアーカイブで配布されているので、まずはそれを実行します。実行されている様子は以下のような画面になります。こちらで表示されているのはただのログなのであまり詳しく見ていく必要は現時点ではりません。



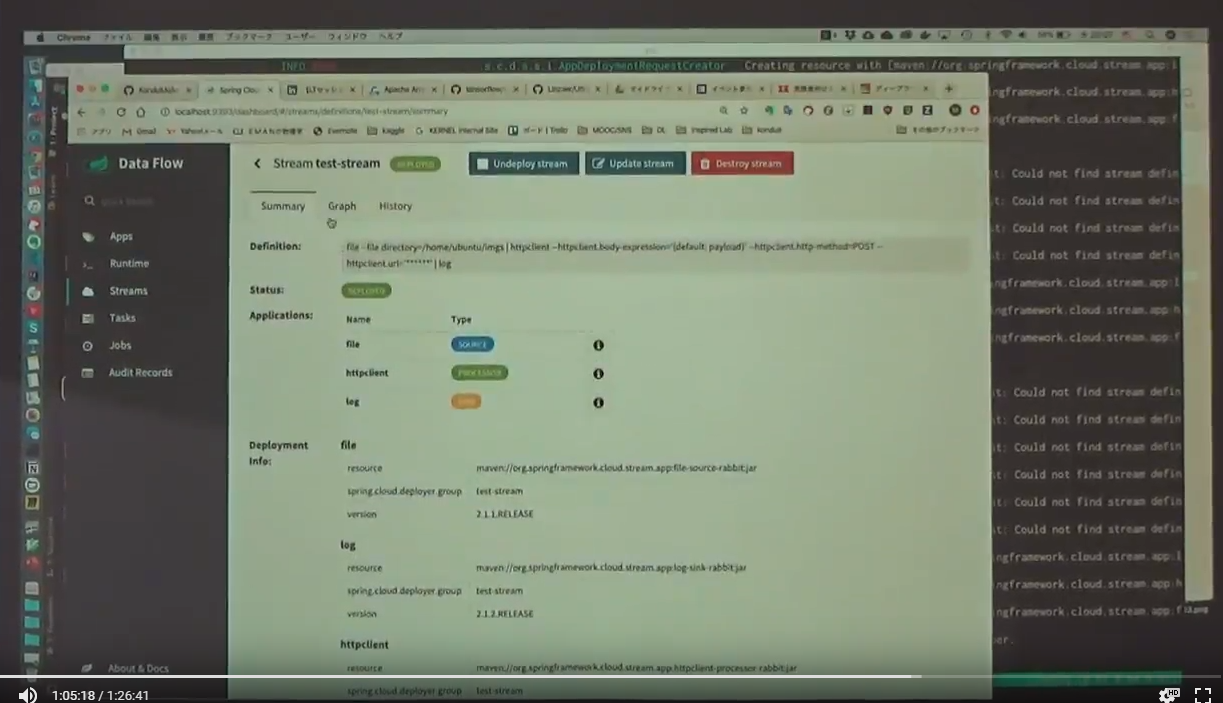
図： Spring Cloud Data Flowの実行状態例

こちらの例ではSpring Cloud Data Flowはローカルホストで実行されています。localhostのポート9393の下のdashboardという場所にアクセスすると下図のようなUIが表示されます。こちらでは左にどのようなブロックが使えるかのリストがあります。



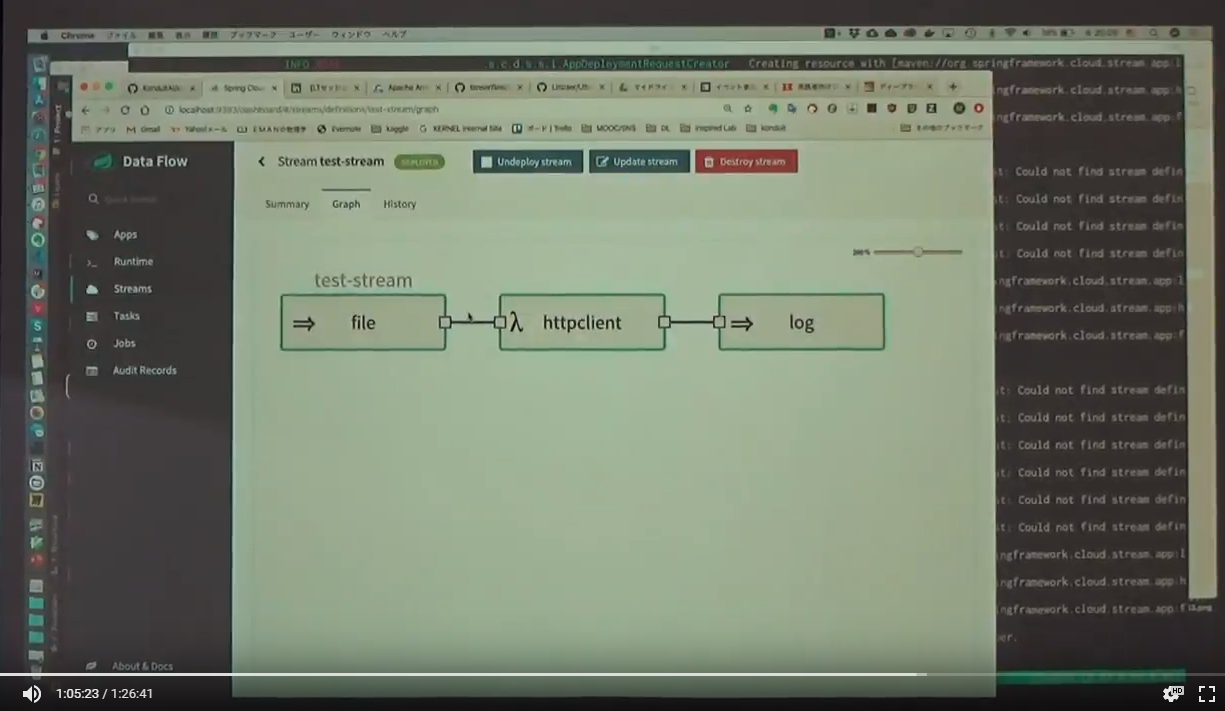
図： localhost:9393/dashboardのUIの表示例

それのさらに左のところにあるタグのうちStreamsというところをクリックすると、下図のように今用意してあるStreamsのリストが見えます。



図： Streamsのリスト例

具体的にどのようなものかというと、上記の図のGraphのタグをクリックすると下図のようにさきほどのデモの説明のときに使用したものと同じブロック図になりますが、図のようにfileがあってhttpclientがあってlogがあるというようなものになっていて、こちらがデプロイされるというような一連のフローのブロックになっています。

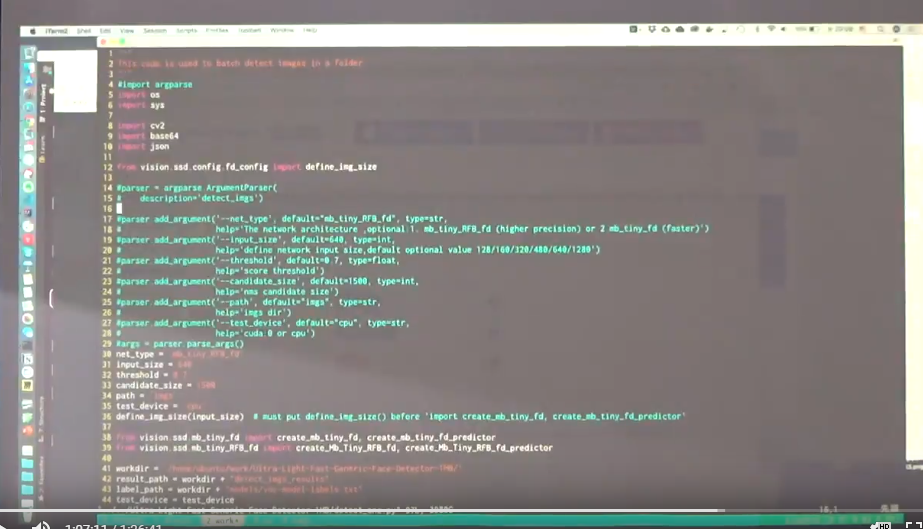


図： 今回のデモで用意したStreamsに関するブロックの例

実際にこちらのプログラムが実行されているかを確かめるのは簡単で、さきほどのStreamsの画面で緑のタグが右側の方にありますが、こちらがdeployedとなっていることを確認いただければ大丈夫です。

こちらのブロックのフローの実際のプログラミングについては、先に説明したドメイン固有言語を用いて、下図のようにfileっていうコンポーネントに対して、引数でディレクトリを指定します。ここではhome/ubuntuディレクトリの下にあるimgsというディレクトリを監視しています。ここに画像が入って来ると、httpclientからpayloadというところに実際の画像が入っているのですが、そのデータをJsonとして投げています。そののちhttpclient.urlで指定されているURLに対してPOSTして、その結果をlogに入れるといった仕組みでこの一連のフローが動作していきます。

ここで実装されている実際のPythonコードは以下のようなもので、こちらは実際にGithubで公開されている推論のコードを持ってきて若干の修正をしただけで大した変更はしていません。



図： 本デモで実装したPythonコードの例

ここでは、base64形式で受け取った画像をNumpyのテンソルに変換してその画像をGithub上ですでに用意されていたpredict関数に入力しているだけです。このpredictで示される関数のコード自身はオリジナルのものをそのまま使っているだけで、こちらの方では何も修正せず採用しています。リターンとして境界ボックスのテンソルであったり、ラベルのテンソルであったり、確率のテンソルというようなものが返ってくるわけですが、その境界ボックスの長さを取って、それを文字列としてdefaultという名前の変数に入力します。defaultに設定する理由は、もともとサーバ上でdefaultという変数をクライアントに返すように設定されているためです。以上が今やっていることの概要となります。



図：サンプルコード例 （こちらは画像ではなく、サンプルコードのスニペット自体を載せるように変更していただけると幸いです）

以下の例では、実際にさきほど指定したSpring Cloud Data Flowが監視している画像が入って来るディレクトリである/home/ubuntu/imgsに新しい画像としてサンプルの人がいっぱい写っている集合写真を1つ追加してみます。画像を追加することで、さきほどのStreamsのフローがバックエンドで実行されて、その画像に写っていた人の顔の数というのをログに出力しています。

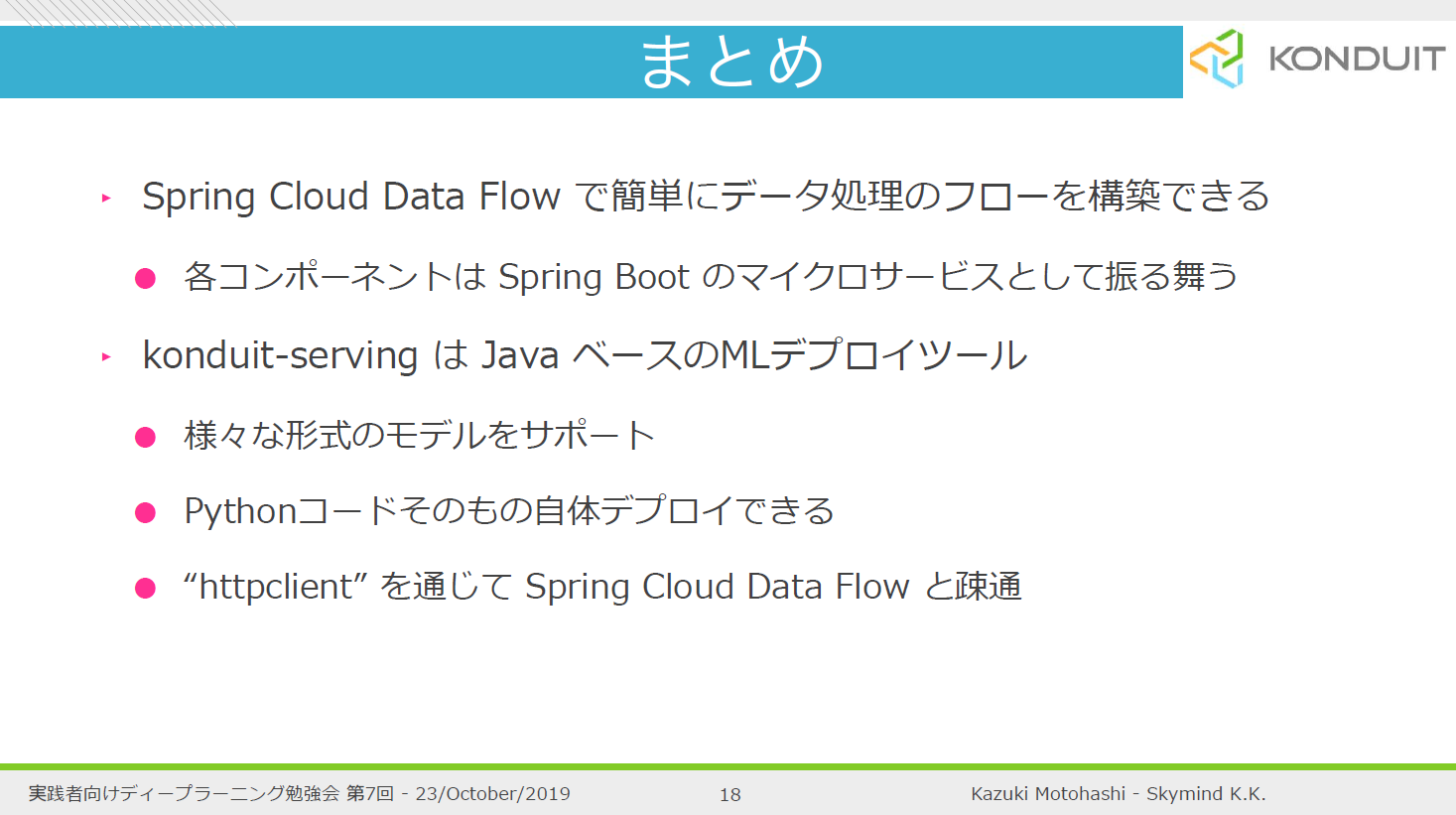
＃＃＃こちらの図については、うまくいったものを載せてください。

上図のようにdefaultに51という数字が読み取れますが、これは今追加した画像に51人の顔が写っているということを認識したという出力結果が示されるわけです。これが非常に簡単なデモの内容となりますが、こちらでお分かりの通り、MLフローをGUIベースのオープンソースのものを使って、さらに公開されているオープンソースのコードを再利用して自分のシステム内で用いることにより、一連のMLライフサイクルを簡単に実装できるのが実感いただけるかと思います。

まとめ

Spring Cloud Data Flowを使用することで簡単にデータ処理のフローを構築することができて、各コンポーネントはSpring Bootのマイクロサービスとして振る舞います。

konduit-servingというツールはJavaベースでさまざまな形式のモデルをサポートし、Pythonコードそのものもデプロイすることができます。また今回のデモでも理解いただける通り、httpclientを通じてSpring Cloud Data Flowと通信することができるというものですので、さらに深く知りたい方は以下のGithubで公開されているので、触ってみてください。



図： 第四章のまとめ

第四章の説明は以上となります。