Computing and Informatics, Vol. 40, 2021, 734–753, doi: 10.31577/cai_2021_4_734

BALTICLSC: LOW-CODE SOFTWARE DEVELOPMENT PLATFORM FOR LARGE SCALE COMPUTATIONS

Krzysztof Marek, Michał Śmiałek, Kamil Rybiński
Radosław Roszczyk, Marek Wdowiak

Warsaw University of Technology
Plac Politechniki 1
00-661 Warszawa, Poland
e-mail: Krzysztof.Marek@pw.edu.pl

Abstract. In modern times, innovation often requires performing complex computations in a short amount of time. However, for many small organisations and freelance innovators, large-scale computations remain beyond reach because of the small accessibility of computation resources and the lack of knowledge required to use them efficiently. The BalticLSC Platform is a software development and computing environment created to address this issue. This paper presents the associated software development process. The platform users can perform advanced computations using ready applications or develop new applications quickly from available components. This can be done using a visual notation called the Computation Application Language (CAL). CAL programs are developed in a dedicated online editor, through selecting and connecting reusable computation modules. If a required module is missing, it can be quickly created by encapsulating code inside a standardised container. The platform's ultimate goal is to relieve the developers from the need to understand the complexity of the distributed parallel computation environment. The platform was implemented in the form of an online software development portal. Validation of the platform consisted in the development of applications and modules by students not experienced in programming. The results of this validation acknowledge the required platform's characteristics.

Keywords: Visual languages, large-scale computing, low-code development

1 INTRODUCTION

In the current rapidly developing world, technology innovation plays a crucial role. In many cases, new discoveries require processing of large amounts of data with the use of sophisticated domain-specific algorithms. Such requirements pose multiple challenges for individual innovators, researchers, small and medium enterprises (SMEs), or research institutions. To use large-scale computations in the development of a new solution, the innovator has to manage the computation resources, have domain-specific knowledge required to solve the problem, and have the programming knowledge to not only implement the solution, but also do it in a way supporting parallelisation and execution at the large scale. On the current market, finding a specialist in a single of these fields poses a challenge, while the knowledge of all of them is required to innovate with the help of large-scale computations.

Over the recent years, large computation resources became widely available for users, mainly due to the popularisation of Cloud Computing and its applications to scientific problem solving [13]. In the past, the researchers had to rely on traditional High-Performance Computing (HPC) [8], focused on single but very powerful homogeneous systems, to perform the required computations. Such an approach posed additional problems for individual innovators. HPC systems were costly and not widely available. Moreover, computation applications were usually written with a specific HPC resource in mind, making the code very difficult, if not impossible, to reuse. A good example of this problem can be found in the results of the SHAPE project [12]. The goal of SHAPE was to promote HPC among Small and Medium Enterprises (SMEs) by providing them with free access to supercomputers and experts capable of developing HPC applications. Such support resulted in successful application of HPC to the problems faced by SME innovators. However, in most cases, the usage of HPC solutions stopped after the external funding has finished. This is due to lack of necessary SME own resources to continue. A possible solution to this problem can be Cloud Computing systems. Similarly to the older idea of Grid Computing [7], such systems allow for better utilisation of computation resources as described by, e.g. Assante et al. [2]. However not only the availability of computation resources poses a threat to new innovations. In his recent article, Pedro Palos-Sanchez describes the lack of knowledge and training in using Cloud Computations, especially among European SMEs [9]. Large-Scale Computations are not only expensive, but also require advanced knowledge to use them.

A possible solution to the described problems can be a Large-Scale Computation Platform that uses a low-code approach. Such approaches use user-friendly visual languages and software development environments to develop applications in various domains [11]. They stem from the model-driven paradigm that consists in defining graph-based languages and developing appropriate language translation mechanisms [5]. In our case, computation domain specialists would gain capabilities to define their computations by using a high-level visual language that hides all

the complexities of the underlying execution environment. This includes complex communication within the execution platform and automatic paralellization of computations. The main goal of this paper is to introduce such a platform, called the BalticLSC and the associated software development process.

2 PLATFORM OVERVIEW

The BalticLSC Computation Platform was created to allow its end-users to perform advanced computations, requiring more than one PC to complete within a few hours, as batch-processed applications. The aim was to require as little programming knowledge as possible, and increase accessibility to computation resources by providing seamless integration of already existing computation resources. To accomplish these goals, the platform provides means to connect computation resource providers, software developers, and computation end-users.

The logical and physical architecture of the Platform is presented in Figure 1. The central physical element of the network is the Master Node. This node hosts the main control software of the Platform. The Frontend component provides the web interface through which the end users can define and execute their applications. The Master Node Backend component is responsible for storing all the computation data, managing Computation Tasks (CT)[1], distributing Computation Jobs (CJ) and managing communication within the Platform through a set of gRPC and REST APIs.

The Platform can be composed of many Cluster Nodes (see again Figure 1). These nodes can be easily registered with the Master Node by the resource providers and then participate in executing computations. Each Cluster Node hosts a standard container orchestration environment (Kubernetes or Docker Swarm). Such nodes can execute CJs as standard containers compliant with the appropriate containerisation standard. Within each Cluster Node, jobs are managed by a dedicated component – the Batch Manager. The Batch Manager receives batches of CJs from the Master Node Backend and requests the creation of appropriate namespaces and jobs within the container orchestration environment. Considering this, it can be noted that connecting a resource to the BalticLSC Platform is relatively easy, especially for regular computation resource administrators. In most cases it just necessitates installing the small Batch Manager component within an already running container orchestration environment. It allows to connect computation clusters of different size and facilitates better utilisation of resources which in many cases are not fully used.

The basic building blocks of any Computation Application (CA) are Computation Modules (CM). These modules are compiled and stored as containers in an external container repository and then executed as appropriate CJs. CM programmers can use dedicated software development kits that facilitate communication with the

[1] See Section 3 for explanation of Computation Tasks and Computation Jobs.

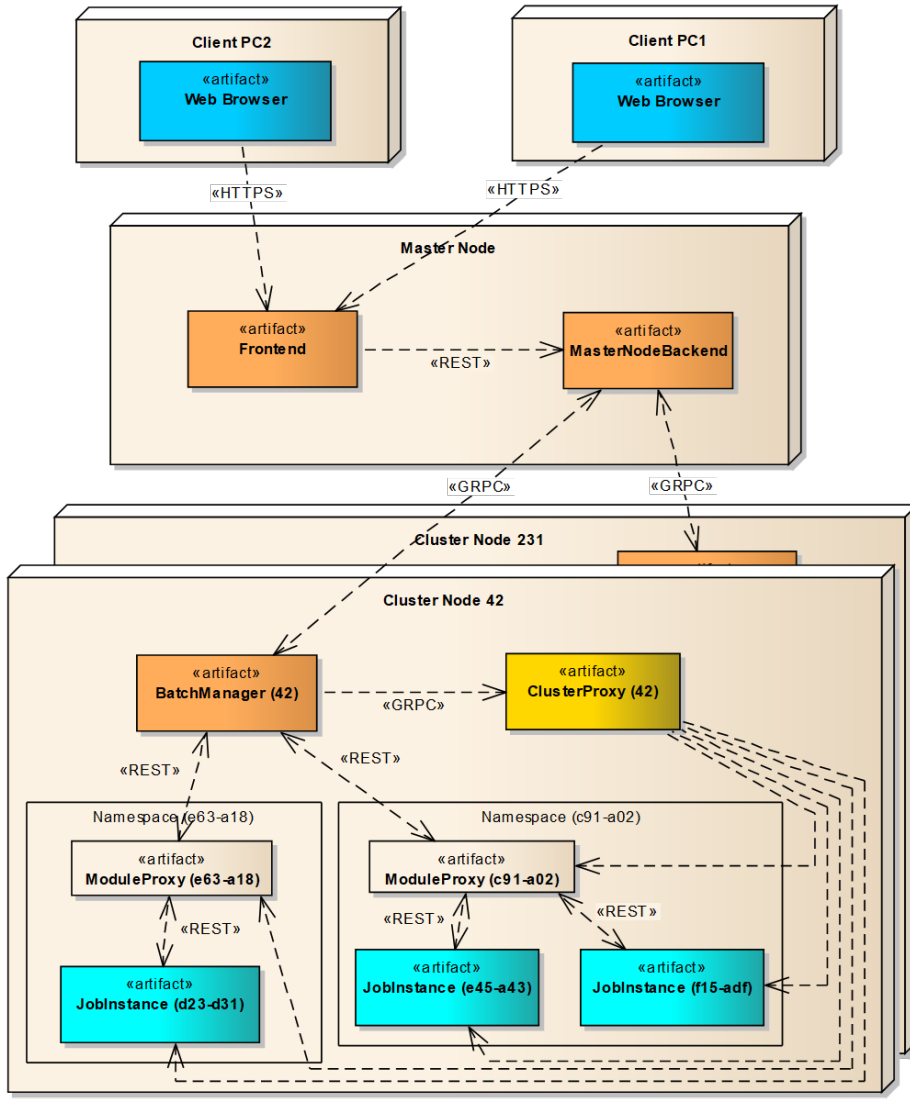

Figure 1. BalticLSC architecture

Batch Manager. This is implemented through a set of proxies that handle REST communication.

The BalticLSC Platform provides an online system that can be used to execute Computation Tasks (CT) based on CAs and to create new applications from CMs. The online system consists of five elements: Application Store, Computation Cockpit, Data Shelf, Development Shelf, and Application Editor.

The Application Store is a place where the platform users can search for available CAs and CMs. The existence of the Application Store allows developers to reuse the CAs and CMs created by others. A monetisation mechanism is planned to be implemented to make a more significant incentive for developers to develop and publish CAs and CMs.

The Computation Cockpit is used by the platform end-users to execute computations. To start some computation using a CA, a CT has to be created. A CT can be started in a weak or a strong mode. The weak mode means the CT can be distributed through assigning its constituent CJs to different Cluster Nodes. In strong mode, all the CJs of a CT will be executed within a single computation node. In such a case, the user can indicate the particular node where the computations should be performed (e.g., depending on its geographical location or its performance characteristics).

After creating a CT, the end-user must provide Data Sets required by the CA to start the computation. Data Sets are definitions of the exact place from where input data can be downloaded or where the CT's output should be uploaded at the end of the computation. The CT will not start until the required Data Sets are provided. In some cases, Data Sets can be provided during the CT execution.

Data Sets are defined by the end-user in a Data Shelf and are universal. They can be used both as input and output, provided that the privileges allow for accessing or uploading the data. To define a Data Set an end-user provides an access path and required credentials to a specific file or folder, for example, located at an FTP server or inside a cloud storage. Once defined, a single Data Set can be reused multiple times between multiple runs of the same application as well as between different applications. Such universality limits the amount of work required to perform computations, because the end-user does not need to provide the access credentials for every CT whenever they are the same.

To create a new CA or a more advanced CM, the developer has to use the Development Shelf. All the owned CAs and CMs can be accessed in the Development Shelf. The owned CMs can be added to the developer's Toolbox and used while editing or creating a new CA. Moreover, the developers can create different versions of CAs and CMs and publish them as appropriate releases. The release mechanism can be used to control multiple versions of the same module. A specific release can be made publicly available or only accessible by its creator.

The CAs present in the Development Shelf can be edited using a dedicated, graphical Application Editor. This editor is web browser-based and allows to develop applications is a special-purpose graphical language called the Computation

Application Language (CAL). We present the language in the next section. The editor works by saving CAL diagrams in real-time, while the diagrams can also be exported as JSON files.

An example workflow of the BalticLSC Platform involving application development and computation execution is presented in Figure 2. The users' work begins in the Development Shelf where they search for the necessary CMs and add them to their toolbox. Next, they can create a new CA and edit it by connecting the selected CMs in the Application Editor. In the editor, they can specify the types of data used by or returned by the CMs. Once the application is ready, the author has to create a new release in the Development Shelf. Then, to run the new CA, the end-user has to add it to the Cockpit in the Application Store. Next, if the application needs data not previously used by the end-user, they need to define new Data Sets in the Data Shelf. Finally, a new CT can be defined in the Computation Cockpit. The computations start once the required Data Sets are selected by the end-user. When the CA is running, the end-user can monitor its progress by checking the status of each CM. If the computation is taking too long or the first results are unsatisfactory, the CA can be manually interrupted.

3 COMPUTATION APPLICATION LANGUAGE

To perform computations on the BalticLSC Platform an end-user has to run a Computation Application (CA) in the form of a single Computation Task (CT). Depending on its parameters, a single CA can solve a specific computation problem, perform analysis of provided data, or perform operations on data. A CA consists of one or more Computation Modules (CM) connected together and described by a low-code visual language called the Computation Application Language (CAL). A single CM is a self-contained piece of software that performs a well-defined computation algorithm. In the runtime environment, CMs are instantiated in the form of Computation Jobs (CJ). In special cases, an entire CA can be used as a CM and participate in developing other CAs. For the purpose of clarity, in this paper, the name Computation Module will be used only to describe a single self-contained module.

Every CA defined in CAL consists of 3 elements: Module Calls, Data Pins, and Data Flows. Their concrete syntax is shown in Figure 3. Module Calls always refer to respective Computation Modules. A Module Call indicates execution of one or more instances of a CM. The actual number if instances depends on the number of data items to be processed, as defined by the associated Data pins (see below).

To perform computations, the CM instances need to be provided with data. In the BalticLSC Platform this data is represented by Data Sets. Access to these Data Sets is provided to the CM instances through so-called Data Tokens passed through Data Pins. Data Pins of type "input" accept Data Tokens that point to Data Sets

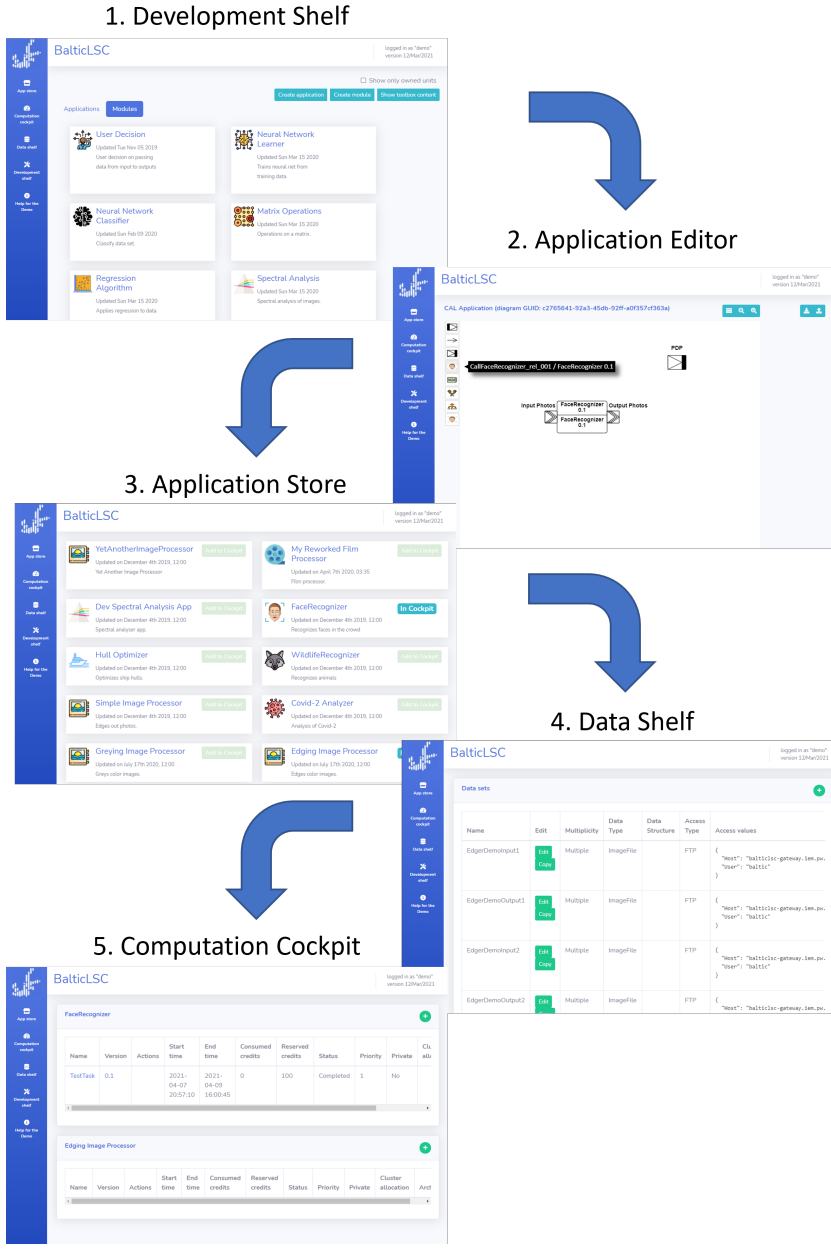

Figure 2. BalticLSC Platform user interface and workflow

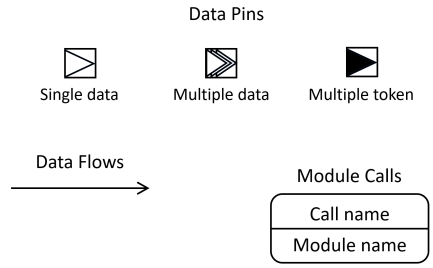

Figure 3. Main syntactic elements of the Computation Application Language

provided by other modules or directly by the user. Data Pins of type "output" deliver Data Tokens that point to Data Sets provided by the current module. CAL distinguishes several types of configurable Data Pins depending on the multiplicity of the data. "Single data" pin is used to represent a single file, a single database table, or other single data container. "Multiple data" pin represents multiple files in the form of a file folder, database schema, or a collection of data containers. The last type of Data Pins is "multiple token", representing a sequence of data items. For example, multiple files sent one after another. "Multiple token" Data Pins allow for batch processing of many individual data items independently and possibly in parallel.

Data Pins can be used in CAL programs in two ways. First, they can be used as standalone elements representing inputs and outputs for the entire Computation Application. A standalone Data Pin is indicated by a vertical black bar before or after the arrow, as shown in Figure 4. The black bar before the arrow indicates that the Data Pin is representing an input to the entire CA. Analogically, the black bar behind the arrow indicates the output of the entire CA.

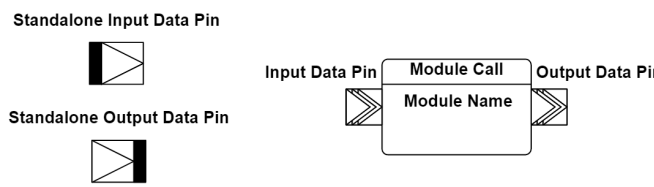

Figure 4. Pins and modules in the Computation Application Language

The second use of Data Pins is in conjunction with Module Calls. In this case, Data Pins are placed on the edges of the appropriate Module Calls. They represent the input and output data of the CM. A specific CM instance starts its operations when it receives Data Tokens on every of its required input Data Pins. When

the module finishes its computations (all or part), it sends Data Tokens to specific output Data Pins.

To define flow of data in CAL programs one needs to connect Data Pins with Data Flows. Data Flows are represented by simple arrows connecting output Data Pins with input Data Pins. This indicates the transfer of Data Tokens within a CAL program (cf. Computation Application). The Data Flow arrows have to start at either output Data Pins of Module Calls or at standalone input Data Pins. Analogically, they have to end at either input Data Pins of Module Calls or at standalone output Data Pins.

4 EXAMPLE CAL APPLICATIONS

The presented notation of CAL seems simple at first sight. However, its semantics allows for constructing even sophisticated parallelisation scenarios. In this section we present some example CAL applications that illustrate basic parallelisation capabilities of CAL.

4.1 Simple Application – Face Recogniser

The simplest valid Computation Application written in CAL has to contain a single Module Call, connected to one input and one output standalone Data Pins. An example of such a CA is shown in Figure 5. This application takes pictures from a Data Set assigned to the input Data Pin ("Input Photos") and processes them inside a CM ("Face Recogniser") instance. This consists in a neural network algorithm detecting and marking peoples' faces on the pictures. After processing all the images, they are uploaded to the Data Set specified by the output Data Pin ("Output Photos").

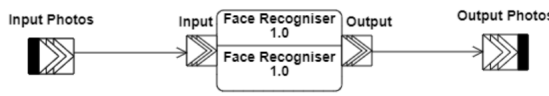

Figure 5. CAL diagram of the Face Recogniser Computation Application

Note that this simple CAL program uses "multiple data" pins. This means, it operates on whole sets of data items (here: pictures). Thus, there is no parallel execution of CM instances. All the pictures are processed by a single instance of "Face Recogniser". In order to make the application "parallelizable" one would need to develop a "Face Recogniser" module that accepts single pictures, i.e., with "single data" pins. In such a case, the BalticLSC execution engine would take individual pictures from the input Data Set and assign each of them to a separate instance of the new "Face Recogniser" module. This would allow for parallel processing of each

of the pictures, where the CM instances could be potentially distributed between various Cluster Nodes.

4.2 Advanced Application – Image Edger

A more advanced CA written in CAL is shown in Figure 6. The presented CA detects the edges of the provided pictures by splitting the image into three colours and analysing them independently. As shown in the diagram, the pictures are taken from the outside repository assigned to the "Data Input" pin. Each of the pictures is processed by an instance of the "Image Channel Separator" module. It splits a picture into one of the three RGB colours. Note that a single token arriving at the "Input Image" pin causes the creation of three tokens – one on each of the output pins ("Channel 1-3"). Each channel is processed by a call to an "Image Edger" CM specific for the given colour. The final step is to merge the outputs of the thee "edgers" into a single image. An instance of the "Image Channel Joiner" module is started when tokens on each of the "chanel" input pins arrive. Note that the execution engine keeps track of the tokens and assures that the "channel" tokens arriving at a particular "Image Channel Joiner" instance relate to the same picture.

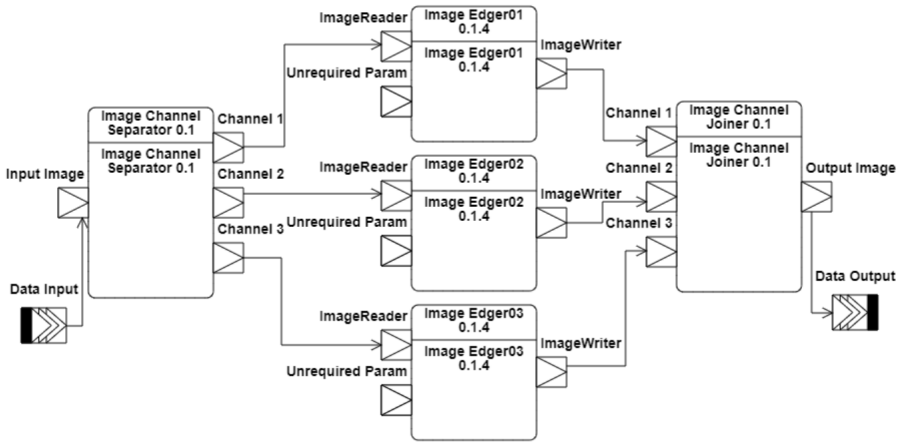

Figure 6. CAL Diagram of Image Edger computation application

The Image Edger application includes several possibilities to parallelise computations. When executing this application, the BalticLSC execution engine takes several pictures from the input Data Set and based on this it creates several instances of the "Image Channel Separator" module. Furthermore, it creates three instances of the "Image Edger" modules per each instance of the separator. Finally, it creates an instance of the "Image Channel Joiner" module per the three instances

of the edgers. For n input pictures, we thus obtain $5n$ module instances running in the system, potentially in parallel.

The edge detection algorithm can be used as standalone but also as a step in more advanced algorithms. Therefore, this CAL program could be easily turned into a module and reused through a Module Call in some other applications.

5 DEVELOPMENT OF COMPUTATION MODULES

The library of Computation Modules forms the backbone of the BalticLSC Platform. Modules encompass various algorithms performing computations in different problem domains. Modules are reusable and can be interfaced with other modules through compatible Data Pins. This allows for building more complex algorithms through combining several module calls in a CAL program.

5.1 Computation Module Lifecycle

In order for a CM to operate within the BalticLSC system, it needs to be made compliant with specific deployment and communication rules. The most basic requirement is that it needs to be compiled and deployed as a Linux Docker container. It also needs to communicate with the BalticLSC Engine through specific REST interfaces (APIs). This allows to operate within the BalticLSC module execution environment that is illustrated in Figure 7. To assist the module development process, example modules in Python and C# are publicly available, together with a dedicated SDK and a developer's manual.

CMs working within the execution environment are called Job Instances. Each instance is executed on a specific Cluster Node and its instantiation and termination is managed through a specific Cluster Manager (using the Kubernetes or the Docker Swarm technology). The instances' lifecycles and passing of Data Token are managed by the Batch Manager component. The Batch Manager is an intermediary between the Job Instances and the central Master Node Backend component. It also instructs the Cluster Manager to start or finish the instances.

To enable proper communication between a CM and the Batch Manager, every CM has to implement a simple REST API (called JobAPI) consisting of two endpoints and has to use another simple REST API provided by the Batch Manager (called TokensAPI). It also has to read appropriate configuration data and access appropriate data stores with the information provided by the Data Tokens.

Computation Module programmers thus have to follow a specific standard lifecycle. The list of steps in the lifecycle is provided below.

1. Read appropriate configuration data and set-up connections with the infrastructure (data stores, API endpoints, etc.).
2. Receive one or more Data Tokens on the JobAPI.

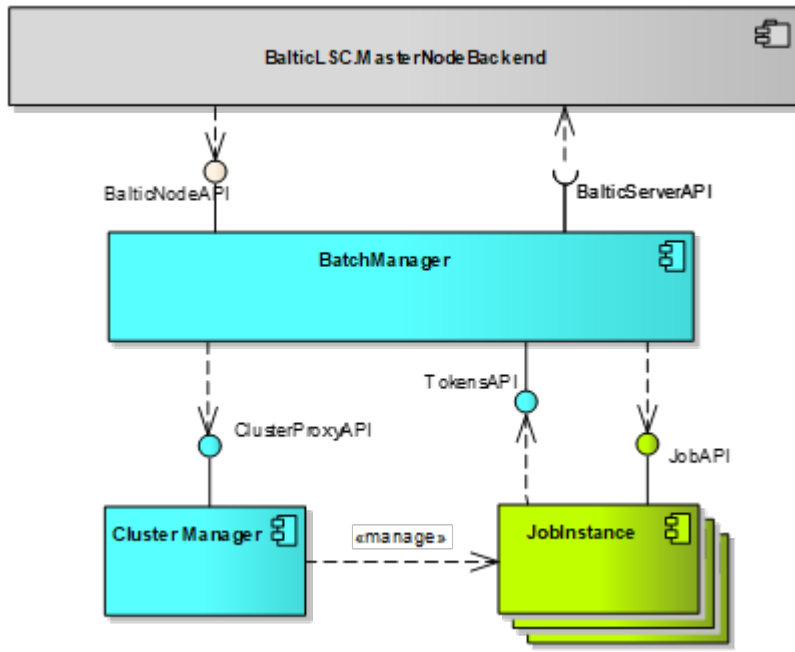

Figure 7. Component structure of the Computation Modules' execution environment

3. Access and start processing Data Sets based on the received tokens and input pin configuration.
4. Update existing or create new Data Sets based on output pin configuration.
5. When finished updating/creating some Data Set, send a Data Token to the TokensAPI.
6. When completed the computation execution, send an acknowledgement message for all the received tokens.
7. Reset all the internal states and prepare for potential processing of a next computation execution.

5.2 Computation Module Configuration

Configuration of a Job Instance is determined through environment variables and configuration files. Every CM is obliged to read several standard environment variables and process a standard pin configuration file. Below we present detailed information on these standard configuration elements. Additional configuration elements can be defined by the module developer if they are required to perform a specific computation.

- SYS_MODULE_INSTANCE_UID – the identifier of the module (Job Instance) granted by the Batch Manager that should be set in all the output tokens;
- SYS_BATCH_MANAGER_TOKEN_ENDPOINT – the address of the PutTokenMessage endpoint;
- SYS_BATCH_MANAGER_ACK_ENDPOINT – the address of the AckTokenMessages endpoint;
- SYS_PIN_CONFIG_FILE_PATH – the path to the pin configuration file, that is generated and provided by the Batch Manager.

To access data, the CM uses the pin configuration file from the SYS_PIN_CONFIG_FILE_PATH variable. Inside the file, an array of Data Pin definitions in the form of a JSON object can be found. Each pin definition consists of several attributes. An example JSON file with two pin configuration has been provided below.

```
[{ "PinName": "Image Folder",
   "PinType": "input",
   "AccessType": "MongoDB",
   "DataMultiplicity": "multiple",
   "TokenMultiplicity": "single",
   "AccessCredential": {
      "User": "someuser",
      "Password": "somepass",
      "Port": "27017",
      "Host": "b-36a1a684-51a8"
   }
},
{  "PinName": "Images",
   "PinType": "output",
   "AccessType": "FTP",
   "DataMultiplicity": "single",
   "TokenMultiplicity": "multiple",
   "AccessCredential": {
     "Host": "ftp.somehost.com",
     "User": "someuser",
     "Password": "somepass"
   },
   "AccessPath": {
     "ResourcePath": "/files/out"
   }
}]
```

5.3 Token Processing by the Computation Module

For the CM to work inside the Platform, it has to receive and process Data Tokens correctly. Data Tokens indicate from where the CM can access input data and they inform the Computation Engine where the data output from the CM has been uploaded. Without correct processing of Data Tokens, the CM would be isolated from the rest of the Platform and would not be able to use full capabilities of the BalticLSC environment. The process is started when an input Data Token message is received at the ProcessTokenMessage endpoint of the JobAPI and handled by the JobController. Following this, the CM checks for the correctness of the token's structure and contents ("CheckToken"). If the token is incorrect, the CM immediately sends a response message "corrupted-token". Further on, the CM checks connections to data stores ("CheckDataConnections"). In case when the data store does not respond and time-out occurs, the module immediately sends a response message "no-response". In case when the data store responds by indicating that the authorization or access to the data path has failed, the module immediately sends a response message "bad-credentials". If the token and data connections are correct, the module starts processing the data contained in the token ("StartDataProcessing"), sets the computation status of the module to "Working" and immediately sends a response message "ok". Normally, the processing workload should be started asynchronously as a separate thread. When processing data, the module can connect to appropriate data stores through AccessCredentials taken from the pin configuration file. During and after finishing data processing, appropriate acknowledgement tokens and output token messages are sent. When the module processes data (cf. "DoProcessData"), it creates some Data Set, usually by accessing (writing to) an appropriate data store, according to the specific Data Pin definition. When the output Data Set is ready, the module sends an output Data Token to the "PutTokenMessage" endpoint of the TokensAPI. The output token message has to contain the appropriate AccessPath data of the data item created by (output from) the module. When the module finishes a complete lifecycle for a single computation algorithm (processes all received input tokens), it sends an "ack-ok" message to the AckTokenMessage endpoint of the TokensAPI. When the module encounters some data processing error (caused by corrupted data etc.), it should send a "failed-data-processing" message to the AckTokenMessages endpoint. Once the error message is received, the computation stops, an appropriate error message is displayed, and the end-user can download the logs from the BalticLSC Platform. The end-user can also manually monitor the status of each of the CMs based on the tokens received and stop the computations at any time.

It is worth noting that the Batch Manager will transform the output tokens received from one CM, into input tokens for the next CM. The tokens will start appropriate further instances of (other) computation modules if necessary, and as defined by the CA. Such approach allows for connecting different CMs inside

a single CA without the need for changes in the code of the CM (provided the type of data and access to them is compatible between the two CMs).

6 VALIDATION OF THE MODULE DEVELOPMENT PROCESS

The presented development process has been initially validated through a project run in academic environment. The aim was to verify the difficulty of developing new CMs by inexperienced platform users. A group of Computer Science students was tasked with developing a new Computation Application using an already trained neural network to detect the emotions of people on the provided images. To accomplish the task, a new CA containing several new CMs had to be implemented. The CAL Diagram of the Emotion Detection Application is presented in Figure 8. First, to allow for Emotion Detection, the faces in the input images had to be detected. Next, a different module, using another pre-trained neural network has to detect characteristic points on previously detected faces. Another CM recognises the mood of the person detected on the picture using the characteristic points on their face. The results are then summarised by creating a JSON file with the detected emotion of each detected face on the image. This is sent to the user together with the images with marked faces. Both the input download and output upload modules were generic CMs available for everyone on the BalticLSC Platform.

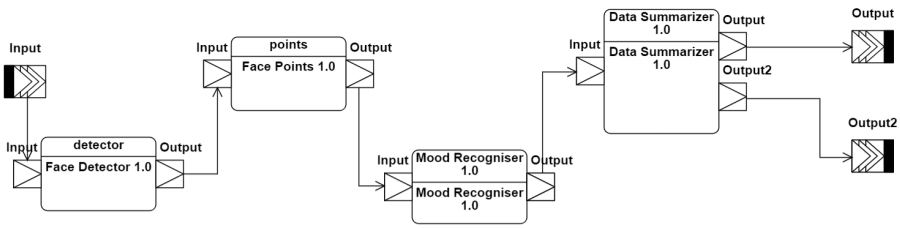

Figure 8. CAL diagram of the Emotion Detection application used during validation

The participants were provided with the CM Development instructions described in the previous section and an example module. They did not have information or any previous experience with the BalticLSC Platform architecture or orchestration solutions (Kubernetes and Docker Swarm) used to manage the CM. During the experiment, the BalticLSC Platform was still under development. Therefore, parts of its documentation and the example module were updated during its duration. The updates were partially based on the feedback gained during the validation process, especially the description of Data Pins in the manual. Participants stated that the introduced changes were very helpful and the updated example module was easier to use and understand.

The biggest inconvenience brought up by the participants was the lack of testing environment for developers. Therefore the new versions of the CMs needed to be manually added to the platform by the administrators. This sometimes took one or two days which significantly slowed down the development process. Since the experiment, the testing environment has been implemented and is available to the developers. At the end of the validation, all the participants were able to produce working CMs which were successfully communicating within the BalticLSC Platform. This was despite they had only basic programming skills and knowledge about REST APIs and containerisation. With the required CMs already implemented, creating the CA (CAL program) was instant for every participant.

7 RELATED WORK

At its core, the BalticLSC Platform is a heterogeneous distributed computing platform. The idea of connecting multiple different machines to form a computational network capable of solving advanced problems is not new. One of the well-known implementations was the Seti@home project [1] using the computation power of private computers selflessly connected to the network via the Internet to search for Extraterrestrial Intelligence. However, this solution was designed with a specific task in mind and was not universal. A good example of a more universal solution in the XtremWeb project [6], a peer-to-peer computation system capable of solving different computation problems. This solutions enable large-scale computations in a distributed environment but are still requiring extensive knowledge and experience to successfully develop a computation application. The solution to this problem can be a visual programing environment.

The idea to use a visual language to ease the development of HPC applications existed among researchers for a long time [14]. Current visual solutions focus on cloud computation [10] but are still based on code generation which makes the language difficult to expand by implementing new functionalities. An alternative to code generation can be the use of containerization, a technology commonly used in cloud computing. The result of such an approach is Kubeflow [3] software streamlining the process of training of neural networks on a Kubernetes [4] orchestration platform. However, Kubernetes was designed with homogeneous computation clusters in mind, to use it optimally on multiple heterogeneous clusters, an additional orchestration strategy is required as described by Zhong and Buyya [15].

8 CONCLUSIONS AND FUTURE WORK

BalticLSC Platform simplifies large-scale computations on two levels. The first one is performing the computations. The end-user can reuse entire applications (CAs) or their parts (CMs) without the need to write any code. They do not have to manually administrate the computation resources on which the computations are

performed. The second area of sipmplification is the development of the CMs. The programers do not need to have an advanced knowladge of distributed and pararell computing, orchestration solution, etc. To develop your own CM, outside of specific domain knowledge, only the basic REST API and containerization knowladge are required.

Therefore, at its core, the BalticLSC Platform can be applied to solving different computation problems and pararellizing them automatically. Areas like finite element method calculation, neural network training, or image processing can benefit from an easy to use computation environment, allowing for solving multiple problems at once, therefore reducing the time to market of a new product.

The solutions and technologies created for the BalticLSC Platform can be used to create domain-specific large-scale computation platforms. Such specialization would allow for easier development of the critical amount of domain-specific CMs, therefore making CA development easier for users without significant programming experience.

REFERENCES

[1] Anderson, D. P.—Cobb, J.—Korpela, E.—Lebofsky, M.—Werthimer, D.: SETI@home: An Experiment in Public-Resource Computing. Communication of the ACM, Vol. 45, 2002, No. 11, pp. 56–61, doi: 10.1145/581571.581573.

[2] Assante, D.—Castro, M.—Hamburg, I.—Martin, S.: The Use of Cloud Computing in SMEs. Procedia Computer Science, Vol. 83, 2016, pp. 1207–1212, doi: 10.1016/j.procs.2016.04.250.

[3] Bisong, E.: Kubeflow and Kubeflow Pipelines. In: Bisong, E.: Building Machine Learning and Deep Learning Models on Google Cloud Platform. A Comprehensive Guide for Beginners. Apress, Berkeley, CA, 2019, pp. 671–685, doi: 10.1007/978-1-4842-4470-8_46.

[4] Burns, B.—Beda, J.—Hightower, K.: Kubernetes: Up and Running: Dive into the Future of Infrastructure. Second Edition. O'Reilly Media, 2019.

[5] Cabot, J.: Positioning of the Low-Code Movement within the Field of Model-Driven Engineering. Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS '20), 2020, Art. No. 76, doi: 10.1145/3417990.3420210.

[6] Cappello, F.—Djilali, S.—Fedak, G.—Herault, T.—Magniette, F.—Néri, V.—Lodygensky, O.: Computing on Large-Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. Future Generation Computer Systems, Vol. 21, 2005, No. 3, pp. 417–437, doi: 10.1016/j.future.2004.04.011.

[7] Foster, I.—Zhao, Y.—Raicu, I.—Lu, S.: Cloud Computing and Grid Computing 360-Degree Compared. 2008 Grid Computing Environments Workshop, IEEE, 2008, pp. 1–10, doi: 10.1109/gce.2008.4738445.

[8] Hager, G.—Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, 2010, doi: 10.1201/ebk1439811924.

[9] Palos-Sanchez, P. R.: Drivers and Barriers of the Cloud Computing in SMEs: The Position of the European Union. Harvard Deusto Business Research, Vol. 6, 2017, No. 2, pp. 116–132, doi: 10.3926/hdbr.125.

[10] Quiroz-Fabián, J. L.—Román-Alonso, G.—Castro-García, M. A.—Buenabad-Chávez, J.—Boukerche, A.—Aguilar-Cornejo, M.: VPPE: A Novel Visual Parallel Programming Environment. International Journal of Parallel Programming, Vol. 47, 2019, No. 5, pp. 1117–1151, doi: 10.1007/s10766-019-00639-w.

[11] Sahay, A.—Indamutsa, A.—Di Ruscio, D.—Pierantonio, A.: Supporting the Understanding and Comparison of Low-Code Development Platforms. 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020, pp. 171–178, doi: 10.1109/seaa51224.2020.00036.

[12] Telib, H.—Cisternino, M.—Ruggiero, V.—Bernard, F.: RAPHI: Rarefied Flow Simulations on Xeon Phi Architecture. Technical Report, SHAPE Project Optimad Engineering srl., 2016.

[13] Vecchiola, C.—Pandey, S.—Buyya, R.: High-Performance Cloud Computing: A View of Scientific Applications. 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, IEEE, 2009, pp. 4–16, doi: 10.1109/i-span.2009.150.

[14] Zhang, D. Q.—Zhang, K.: A Visual Programming Environment for Distributed Systems. Proceedings of Symposium on Visual Languages, IEEE, 1995, pp. 310–317, doi: 10.1109/VL.1995.520824.

[15] Zhong, Z.—Buyya, R.: A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources. ACM Transactions on Internet Technology (TOIT), Vol. 20, 2020, No. 2, Art. No. 15, pp. 1–24, doi: 10.1145/3378447.

Krzysztof Marek is Assistant at the Institute of Control and Industrial Electronics of the Warsaw University of Technology. He is currently Ph.D. candidate. His research interests include agile software development methods, business process modelling, requirements engineering, large-scale computations and software engineering metrics.

Michał Śmiałek is Professor of software engineering at the Warsaw University of Technology. He obtained his habilitation (higher doctorate) degree in informatics from the Warsaw Military University and graduated from the Warsaw University of Technology (M.Sc. and Ph.D.) and the University of Sheffield (M.Sc.). Since 1991 he has worked in the industry as Software Developer, Project Manager and Professional Coach. His current research interests include model-driven software development, requirements engineering, software reuse, software language engineering and large-scale computing. He published and edited several books and over 100 papers in various journals and conference proceedings. He coordinated two European-level research projects, chaired several international conferences and reviewed for major computer science journals.

Kamil Rybiński is Adjunct Professor at the Institute of the Theory of Electrical Engineering and Applied Informatics of the Warsaw University of Technology. He obtained his Ph.D. in software engineering from the Faculty of Electrical Engineering at the same university. His research interests includes requirements engineering, model-driven software development and knowledge representation.

Radosław Roszczyk specializes in biomedical image analysis and processing, machine learning, data analysis, and distributed systems. He has experience in developing new methods and their application in a wide range of scientific and technical fields, including biomedical engineering, horticulture, and measurement systems. He participated in international and national research projects. For nearly 20 years, he worked in global corporations, and currently, he is an employee of the Faculty of Electrical Engineering at the Warsaw University of Technology. He is Member of IEEE and ISHS.

Marek Wdowiak has focused his research field on software engineering and image processing. He conducted research centres around issues, such as segmentation and quantitative analysis of microscopic images, mathematical morphology in image filtration, machine learning.