Constructing a unified platform for the scalable, reliable, robust, and efficient development and deployment of ECCI applications.

BY LUHUI WANG, CONG ZHAO, SHUSEN YANG, XINYU YANG, AND JULIE MCCANN

ACE: Toward Application-Centric, Edge-Cloud, Collaborative Intelligence

IN RECENT YEARS, machine learning (ML), especially deep learning (DL), has been applied to various domains—for example, computer vision, speech recognition, and video analytics. Emerging intelligent applications (IAs), such as image classification based on deep convolutional neural networks (CNNs), traffic-flow prediction based on deep recurrent neural networks (RNNs), and game development based on deep generative adversarial networks (GANs) are demonstrating superior performance in terms of accuracy and latency. Such performance, however, requires tremendous computation and network resources to deal with the increasing size of ML/DL models and the proliferation of vast amounts of training data.

Cloud computing is indisputably attractive to IA developers as the pre-dominating high-performance computing (HPC) paradigm. Cloud providers typically offer services such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) to facilitate application implementation—resources such as high-performance computation, massive elastic storage, and reliable network services are allocated according to user requirements. Intuitively, mainstream IAs are deployed on the cloud to leverage centralized resources for computationally intensive artificial intelligence (AI) tasks, such as data processing, ML/DL model training, and inference. For instance, the distributed training of AlphaGo is a typical cloud intelligence (CI) representative.

However, novel challenges to CI emerge when modern IAs rapidly proliferate and need to be in production in...
practice, where high end-to-end service latency, high network-bandwidth overhead, and severe privacy leakage threats are among the most critical ones. Instead of concentrating on the cloud, increasing efforts attempt to exploit heterogeneous resources distributed at the network edge to address such issues. Some IAs offload DL tasks to edge servers (for example, Nvidia’s Jetson TX2 board) for privacy preservation and timely responses. Such an edge offloading of relatively simple AI tasks, or edge intelligence (EI), alleviates the controversy between broadened requirements of modern IAs and the conventional CI paradigm.

The rapid development of EI and corresponding prototypes demonstrates that, due to edge devices’ heterogeneous resource constraints, the cloud is still critical to modern production-level IAs with multi-faceted performance requirements. Increasingly, IA developers have begun focusing on efficiently leveraging edge resources under cloud coordination to collaboratively conduct AI tasks with optimized performance, or edge-cloud collaborative intelligence (ECCI). ECCI relies on pivotal interdisciplinary technologies of cloud and edge computing (supporting ECCI infrastructure and runtime) and ML/DL-based AI (introducing rich IA workloads).

Existing ECCI applications (for example, HOLMES for healthcare, EdgeRec for e-commerce, SurveilEdge for urban surveillance, and general solutions such as CLIO and SPINN) are individually developed and deployed either by academic researchers or industrial communities, where both the application design and system implementation are highly developer dependent and scenario specific. For example, SurveilEdge is a typical ECCI application for real-time, intelligent, urban surveillance video query. In its prototypical implementation, developers depend on relatively higher edge computation capabilities (that is, x86 PCs) to support system scaling without subtly designing an ECCI infrastructure management scheme. For the ease of implementation, they hard-code the load-balancing policy with the video query workload for latency reduction. Additionally, to achieve intelligent video query, the entire solution is specifically designed to support CNN training and inference workloads, where dedicated service links (for example, message service links) among all application components are individually configured to achieve edge-cloud collaborations. However, without impacting application performance, such developer-dependent design and implementation impedes others to migrate the application to general ECC infrastructures (for example, resource-constrained industrial IoTs) or to pursue customizable performance optimizations (for instance, joint optimization of latency and bandwidth consumption). Moreover, if others want to adopt SurveilEdge (or other existing applications) as the backbone for other applications driven by different DL models and deployed at different infrastructures, corresponding DL runtimes and different ECC services must be thoroughly designed and implemented by the adopters themselves. Such a non-generic manner is severely hindering the proliferation of production-level ECCI applications.

Therefore, for the cost-efficient implementation of high-performance, production-level ECCI applications, it is necessary to construct a unified platform to handle both ever-increasing edge and cloud resources and emerging IA workloads with increasing scale and complexity. Particularly, to construct such a platform, the following four challenges must be explicitly addressed:

Support for unified management of hierarchical and heterogeneous infrastructures. The efficient implementation of ECCI applications requires unified management of not only infrastructures offered by traditional centralized cloud providers but also heterogeneous computation, storage, and network resources geographically dispersed at the edge. Developing and deploying ECCI-application components on edge devices is extremely inefficient due to the lack of a unified platform. Furthermore, it is infeasible to directly migrate IaaS and PaaS technologies in cloud computing to the management of inherently distributed edge resources.

Support for user-transparent ECC services. ECCI application developers require services providing user-transparent edge-edge and edge-cloud collaborations. In most cases, components
of existing ECCI applications are independently deployed on edge nodes, only interacting through services deployed on the cloud. This increases both bandwidth cost and response latency. Few existing edge services (for example, Dapr\textsuperscript{a}) can improve edge autonomy and application performance to a certain extent. However, due to the lack of links between edge and cloud services, they cannot provide user-transparent, collaborative services to developers.

**Support for complex IA workloads.** Efficient ECCI application implementations require comprehensive system-level supports to complex IA workloads, such as ML/DL model training and inference, which cannot be provided by existing cloud- and edge-computing platforms. For instance, in edge-computing systems for IoT data processing, the message-driven communication solution for transmitting kilobyte-level sensor data cannot effectively handle the transmission of DL models as large as hundreds of megabytes. Moreover, most existing distributed ML/DL solutions are designed for datacenter networks with high bandwidth and low transmission latency. Such methods are inefficient in ECC systems with inherent constraints, such as prolonged and unstable end-to-end (E2E) communication latency.

**Support for unified optimization of ECCI applications.** Unified performance optimization mechanisms are important to efficient ECCI application implementations. For most existing edge-computing applications, resource-utilization efficiency highly depends on the developer’s design, where effective optimizations require a profound understanding of system architectures and optimization theories.\textsuperscript{14} For existing ECCI applications, except for the multi-component development and cross-device deployment of inherently complex IA workloads, developers must also deal with the overall performance optimization across ECC infrastructures by themselves, not to mention the difficulties in application debugging, monitoring, and profiling caused by the distributed and heterogeneous environment. Such a requirement is quite challenging to not only developers of emerging ECCI applications, but also to those who want to migrate existing IAs to ECC infrastructures.

**ECCI Application Patterns**

Currently, there exists no commonly accepted abstraction of general ECCI application patterns, which are critical to improving the efficiency of ECCI application development and deployment. As the foundation of the unified platform, considering the subject of different application tasks, we extract four common patterns: ECC processing, ECC training, ECC inference, and hybrid collaboration.

**ECC processing** of data is the basis of various ECCI applications. Collaborative data-processing applications are often built as pipelines or *directed acyclic graphs* (DAGs). For example, the Steel framework\textsuperscript{13} deploys a streaming analytics pipeline of different data-processing tasks, such as filtering, anomaly detection, and storage, for ECC IoT anomaly-detection applications.

**ECC training** refers to conducting ML/DL model training based on edge-cloud collaborations. Unlike ECC processing, ECC training requires complex interactive and iterative data and control flows between edges and the cloud—for example, training data, model, and hyper-parameter exchanges. For instance, federated learning (FL) is a typical ECC training application that conducts ML training across devices to protect data privacy (for example, Gboard Mobile Keyboard\textsuperscript{11} and Apple QuickType Keyboard\textsuperscript{12}) and to bridge data silos (for example, model training for bank fraud detection\textsuperscript{13}).

**ECC inference** focuses on ML/DL model inference, where only forward propagation is conducted. ECC inference is generally achieved either through intra-model or inter-model collaborations. In intra-model solutions, a single DL model is decomposed, by layers, into two parts (neural network partitioning) deployed at edges and the cloud respectively for collaborative inference—for example, Neurosurgeon,\textsuperscript{19} SPINN,\textsuperscript{22} and JointDNN.\textsuperscript{3} In inter-model versions, however, multiple DL models with different functionality or performance are deployed at edges and the cloud respectively for collaborative inference—for example, VideoEdge\textsuperscript{18} and SurveilEdge.\textsuperscript{41}

**Hybrid collaboration** combines at least two of the three ECCI application patterns mentioned or integrates additional CI/EI capabilities into ECCI applications. For example, ShadowTutor\textsuperscript{a} enables robust HD video semantic segmentation with significant throughput improvement and network data-transmission reduction. Here, cloud servers conduct both the inference of the heavy and general ‘teacher’ model and the training of the lightweight ‘student’ model. Mobile edge devices conduct the ‘student’ model inference.

**ECCI Platform Design Principles**

In this article, we aim to construct a unified platform for the scalable, reliable, robust, and efficient development and deployment of ECCI applications. Doing so requires the efficient management of heterogeneous ECC infrastructures, user-transparent ECC services, and customizable performance optimizations. The desired platform should be treated as *ECCI-as-a-Service* (ECCIaaS), like the concept of Machine Learning-as-a-Service (MLaaS). Specifically, we extract five essential design principles:

**Principle One: An instance of an ECCI application should be an integrated entity that can be managed in a scalable way.** This principle requires the unified management of typical edge and cloud infrastructures, including hardware nodes, such as edge gateways; clusters, such as Kubernetes;\textsuperscript{b} virtual machines; and third-party cloud services, such as Azure IoT Hub.\textsuperscript{34} Any operation of ECCI applications (for example, deployment and monitoring) should be carried out on large-scale collaborative infrastructures organized as a unity. ECCI applications should be able to provide continuously available services when the infrastructures are scaled or upgraded.

**Principle Two: ECCI application components at edges and the cloud should be able to operate in both collaborative and autonomous manners.** Unlike the datacenter network on the Cloud, the edge-cloud network has limited capabilities (for example, bandwidth), and may perform unstably. While supporting collaborations with

\textsuperscript{a} See the Dapr Community website at https://dapr.io/

\textsuperscript{b} See the Kubernetes Community website at https://kubernetes.io/
the cloud, edges should be able to cache data and provide partial services autonomously to mitigate the impact of network partitioning.

Principle Three: Orchestration is essential to ECCI applications. Except for edge-cloud separations, modularized ECCI application components have specific requirements of computation and storage resources as well as deployment locations. Moreover, there can be multiple applications co-located at the same infrastructure. Therefore, component orchestration is necessary to ensure that all application resource and user requirements can be satisfied.

Principle Four: Provide in-app control of ECCI applications. In most cases, offloading computation to edges may not directly improve application performance. Here, in-app control optimization has been demonstrated to be effective in various aspects such as bandwidth saving and E2E latency reduction, which should be seriously considered for application performance enhancement.

Principle Five: Support as many types of ECCI application workloads as possible. ECCI application scenarios are ever-increasing, such as federated model training and ECC model inference. It is essential for the platform to support common application patterns and services, facilitating efficient development and deployment of a broadened spectrum of ECCI applications.

**Application-Centric ECCI Platform**

Driven by the five principles, the explicit design of our Application-Centric ECCI (ACE) platform is as follows.

**Overview.** In Figure 1, we illustrate the general ECCI application development and deployment procedure based on ACE. For application developers, this procedure comprises three major phases: user registration, application development, and application deployment.

In the **user registration** phase, any ECCI application developer can register at ACE as a platform user. The user first requests the registration of an ECC infrastructure at ACE and registers all edge and cloud nodes to form an infrastructure according to operational instructions replied by ACE. Here, a node can be either a physical device or a virtual service, such as an edge gateway, a cloud server, a private or public cloud, and so on. The user can also choose to deploy different resource-level services based on service components provided by ACE on the infrastructure, which can be shared among all the user’s ECCI applications.

Then, in the **application development** phase, the user implements applications in a modularized manner. Specifically, for each application, different components are separated according to user-defined business logic or functionality. Meanwhile, requested by ACE, the user deliberately decouples application control flows with workload flows for collaboration optimization and component reuse. All components are then implemented using the ACE software development kits (SDKs) and encapsulated into containers that can be deployed on edge or cloud according to a component’s resource and user requirements. For each application, the user constructs a topology file describing component relations and resource and user requirements of each component. All component images and corresponding topology files are then submitted to ACE.

Finally, in the **application deployment** phase, ACE determines a deployment plan for all components of a specific application according to the topology file, guaranteeing that all resource and user requirements are satisfied. According to the plan, the application can be deployed on the user’s ECC infrastructure through ACE. All deployed applications are continuously monitored by ACE for maintenance, and corresponding users can upgrade, monitor, and remove their applications at any time.

To achieve the procedure above, we construct our ACE platform in a hierarchical manner with three layers—the platform layer, resource layer, and application layer. The general architecture of ACE is illustrated in Figure 1. Details of each layer are as follows:

**Platform layer.** This layer manages the ACE platform, all registered users, and user nodes and applications. It also offers platform-level services for...
Developing and deploying ECCI-application components on edge devices is extremely inefficient due to the lack of a unified platform.

users and their applications.

Platform management. Our platform-layer manager comprises a controller, orchestrator, API server, pub/sub service, monitoring service, and user interfaces:

The controller manages platform users, their nodes, and their applications—for example, registers and deletes users, shields failed nodes, and controls node component deployment.

The orchestrator determines a deployment plan for all components of each application based on the topology file, ensuring that resource (for example, computing) and user (for instance, location) requirements of all components are satisfied.

The API server provides uniform APIs for querying and manipulating the status of ACE entities (for example, users, nodes, applications) to other platform manager components (for instance, orchestrator, controller).

The pub/sub service provides a bi-directional data transmission channel between ACE and user nodes and applications (for example, delivering deployment instructions from the controller to user nodes).

The monitoring service collects the status, performance metrics, and runtime logs of ACE, user nodes, and applications.

The user interfaces enhance ACE’s user-friendliness with a command-line interface (CLI) and web-based dashboard. For example, the dashboard with a drag-and-drop visual application editor can be used for handy application development.

Platform-level services. Platform-level services are not ACE’s internal features. They can be implemented as requested to improve the efficiency of ECCI application deployment and deployment based on ACE.

Two typical examples include:

- Image registry hosts ACE-provided images (for example, controller, orchestrator), generic runtime images (for instance, Python runtime), and user-provided customized application images.

- Validation testbed allows users to develop, debug, and monitor ECCI applications on an SDN-based application validation testbed. For example, the impact of edge-cloud channel dynamics (for example, bandwidth, delay, jitter) on the testbed can help users understand the actual performance of an ECCI application in real-world networks.

Resource layer. This layer manages the ECC infrastructure of each user. It also provides resource-level services shared among applications deployed on the same infrastructure.

Infrastructure organization. Considering Principles One and Two, ACE requires all nodes of each user to be organized as several edge clouds (ECs) and one central cloud (CC) to host scalable ECCI applications and to enable autonomous operation of edge components. All edge nodes are divided into several groups according to a specific user’s preferences—for example, in terms of nodes’ geographical locations or resources. Each group is organized as an EC, serving all end nodes (for instance, IoT sensors and cameras) that can access the EC through local area network (LAN). All cloud nodes of the user are organized as a single CC, and it can interact with each EC through wide area network (WAN). For each EC and the CC, internal nodes are organized as a cluster (similar to Kubernetes ideally or a node pool for simple implementation).

Treating each EC and the CC as a resource-level operational unit allows ACE to effectively manage the infrastructure and deploy applications on such an infrastructure without considering the explicit management of potentially massive underlay nodes. Moreover, when there is no cloud coordination caused by either CC or edge-cloud communication failure, each EC as a cluster remains (partially) functional, enabling local area collaborations among edge components based on corresponding edge services.

Receiving the user’s registration request, ACE assigns a unique infrastructure ID to the user and establishes a node information record for infrastructure organization. Meanwhile, ACE assigns a unique second-layer ID affiliated with the infrastructure ID to each EC and the CC claimed by the user, where corresponding node registration instructions are automatically generated. Replied from ACE, such instructions are executed by the user on nodes. An agent is deployed on each node, informing ACE of the node infor-
mation and the EC or CC to which the node belongs. ACE assigns a unique third-layer ID affiliated with the EC’s or CC’s ID to each node. The agent is also used for application deployment and application status collection.

Resource-level services. ECCI applications with the typical patterns previously discussed commonly require essential services such as small/big packet communication and data caching/storage. In a specific ECC infrastructure, existing services supporting ECCI applications are conventionally deployed on both ECs and the CC, serving EC and CC clients (that is, application components) respectively to ensure the autonomy of ECs. Each service is accessible to all applications deployed on the same infrastructure. However, due to the lack of links between edge and cloud services, conventional services require application developers to handle complex edge-cloud interactions. Treating conventional message services for small packet communication as an example, as shown in Figure 2, for edge-cloud unicast communications, all EC clients must directly access the message service at CC (Figure 2, Step 1) to communicate with CC clients. Here, the developer must handle the CC message service authorization to each EC client individually, which is quite expensive for large-scale ECCI applications.

Considering Principle Five, to facilitate efficient application development, ACE prefers to provide E2E resource-level services with unified interfaces to EC and CC clients, respectively. Therefore, long-lasting links between services on ECs and the CC must be established. Some conventional services support the direct establishment of such links (for example, service bridging for the message service). Specifically, as shown in Figure 2, ACE implements a resource-level message service, where the long-lasting link between EC and CC message services (Figure 2, Step 2) is established using MQTT topic-bridging. Here, edge-cloud interactions are conducted by an ACE-provided SDK, and each client only needs to interact with its local service with a dedicated interface. For other services, directly establishing long-lasting links is expensive or even infeasible. For example, the link between edge and cloud file services could be established using file synchronization, which in turn supports the direct establishment of these links (for example, service bridging for the file service). Such supports from ACE enable users to achieve modularized development for ACE-organized ECCI infrastructures. For coding, ACE provides the SDKs with access to resource-level services for application components and the user interface to access the essential integrated development environment (IDE). For building, ACE provides the image registry to enable efficient image management and distribution. For testing, ACE provides the validation testbed for applications with advanced architectures (for instance, large-scale components with complex topologies). For deployment, ACE provides the orchestrator and the controller for automatic deployment. For monitoring, ACE provides the monitoring service collecting the status of application components and nodes where they are deployed. Such supports from ACE enable users to develop and deploy basic ECCI applications efficiently. For applications with specific performance requirements (for example, the minimal E2E latency) or with advanced architectures (for instance, large-scale components with complex topologies), ACE provides two extra supports: reusable development and deployment automation.

Reusable development. Considering Principles Four and Five, ACE requires developers to decouple and separate...
control and workload planes of all application components. The control plane conducts in-app control operations, component monitoring, and policy execution—for example, choosing the best partition point for intra-model inference solutions.\textsuperscript{2,9} The workload plane conducts computation, storage, and transmission instructed by the control plane—for example, deep feature compression module\textsuperscript{1} or hybrid collaboration pipeline for data processing and inference tasks.\textsuperscript{9} Such a separation allows ACE to construct a reusable in-app controller, enabling developers to concentrate on implementing ECCI workloads and efficiently contribute to the ACE-based ECCI ecosystem.

For the reusable in-app controller, ACE constructs a series of general in-app control operations (for example, start, filter, aggregate, and terminate), component monitoring operations, and a basic control policy. Determined by the ECC infrastructure, the controller is constructed at the resource level in an ECC manner. The CC controller conducts global coordination-related operations, and the EC controller coordinates components within the EC. Resource-level services support interactions between CC and EC controllers. For applications with specific performance requirements, developers can inherit the general in-app controller and override optimization methods as advanced control policies—for example, the rate control-based optimal edge-cloud bandwidth allocation.\textsuperscript{2}

Deployment automation. Considering Principle One, ACE needs to support efficient application deployment regardless of topology complexity and infrastructure scale. To achieve this, ACE constructs an automatic application deployment method only requiring the application topology file containing information such as application specification, component clarifications, parameters, relations, and deployment requirements. Such a manner prevents users from handling complex component-infrastructure mapping. Specifically, to deploy an application, the user submits the topology file through the user interface to ACE and triggers the orchestration process. According to component deployment requirements, the ACE platform-layer orchestrator binds each component with specific nodes in the infrastructure and resource-level services required, generating the deployment plan. When the user triggers the deployment process, the ACE platform-layer controller generates the instruction to deploy each component instance on the corresponding node according to the deployment plan and sends the instruction to the node agent for execution. Note that users can manage applications (for instance, update and delete) by modifying the topology file. For example, for updates, such as submitting an updated topology file, the user can trigger a thorough update—that is, ACE deletes the previous application and repeats the entire deployment process. An incremental update can also be triggered; ACE automatically deploys updated components according to the new topology file.

How It Works: Intelligent Video Queries Using ACE
To validate our platform in supporting efficient and high-performing ECCI application development and deployment, we first present the entire development and deployment process of an intelligent video-query application based on ACE, then compare the performance of the application implemented with ACE, CI, and EI, respectively.

Application development and deployment. Video queries\textsuperscript{24,41} is one of the killer ECCI applications. To fulfill latency-sensitive, user-specific video query requests (for example, query about the existence of a type of object in video streams from a geographic area), it generally uses edge and cloud resources to retrieve targeted objects from the video streams with a proper tradeoff between query accuracy and response latency under practical edge-cloud bandwidth limitations. In an earlier section, we developed and deployed a video query application (based on Wang et al.\textsuperscript{41}) using ACE.

User registration. As an ACE user, we first mounted all our nodes and organized our ECC infrastructure instructed by ACE. Our infrastructure comprised a CC (one node—a GPU workstation) and three ECs (each with four nodes—an x86 Mini PC and three Raspberry Pis). Figure 3 provides node details. For each EC, all edge nodes are connected to an individual 100Mbps WLAN. Each EC connected to CC via WAN (campus network) with software-limited bandwidth (20Mbps uplink and 40Mbps downlink) and one-way delay (0ms and 50ms as ideal and

---

**Figure 3. ACE-based intelligent video query workflow.**

![Diagram of ACE-based intelligent video query workflow.](image-url)
Figure 4. Automatic application deployment.

**Application Topology**

| Topology: videoquery |
|----------------------|
| **Containers:** |
| - name: ec1-rpi1 |
| - name: ec1-rpi2 |
| - name: ec1-rpi3 |
| **Connections:** |
| - lic |
| - eoc |
| - coc |
| **Resources:** |
| - cpu: 2 |
| - mem: 1000000000 |
| **Labels:** |
| - edge |
| - camera |
| - eoc |
| - lic |
| - rs |
| - coc |
| - gic |

**Deployment Plan**

| Application: videoquery-001 |
|-----------------------------|
| **Containers:** |
| - name: ec1-rpi1 |
| - name: ec1-rpi2 |
| - name: ec1-rpi3 |
| **Parameters:** |
| - INTERVAL: 0.5 |
| - connections: |
| - lic |
| - eoc |
| - coc |
| **Resources:** |
| - cpu_count: 2 |
| - mem_limit: 1000000000 |

**Deployment Instruction**

- **Version:** 3.7
- **Services:**
  - videoquery-001:
    - image: <od-repository-path>
    - parameters: |
      - connections: |
      - resources: |
    - cpu_count: 2 |
      mem_limit: 1000000000

**Figure 4, Automatic application deployment.**

practical networks, respectively). Let each Raspberry Pi receive the real-time video stream from a surveillance camera. We deployed the resource-level message service on the infrastructure.

**Application development.** Our application aimed to fulfill user-specific video query requests accurately and rapidly through edge-cloud collaborations under practical network limitations—that is, bandwidth and delay. We developed the following components: Data Generator (DG), providing the real-time video stream to the edge node; Object Detector (OD), rapidly extracting video frame crops potentially containing moving objects from the video stream; Edge Object Classifier (EOC), conducting lightweight, query-specific binary object classification; Cloud Object Classifier (COC), conducting accurate multi-class object classification; In-app Controller (IC), executing the control policy; and Result Storage (RS), saving all query results. OD on edge nodes was implemented using frame differencing (that is, cropping regions with salient pixel differences across frames) instead of an accurate but complex object detector, such as YOLOv3, for rapid crop extraction on resource-limited edge nodes. COC on CC was a ResNet152 pre-trained on ImageNet ILSVRC15 (4.49% Top-5 error rate). EOC was a MobileNetV2 rapidly trained on the fly by CC whenever there were user-specific queries. To form its query-specific training set, video frame crops containing different classes of objects were extracted on CC by a YOLOv3 pre-trained on COCO (57.9% mAP measured at 0.5 IOU) from historical video data uploaded by cameras at (or nearby) the queried area at leisure time, then labelled by COC. The trained EOC (training details are in Wang et al.41) was then deployed on edge nodes in the queried area. We used real video clips from YouTube Live (30 fps, 1920 × 1080 resolution, various durations) as historical video data and real-time video streams to query. For a motorcycle query task, EOC’s training set had 14,000 crops extracted from clips (170 hours total duration) from 14 surveillance cameras at or nearby the queried area—that is, historical video data. Another 6,433 ‘motorcycle’ and 68,749 ‘non-motorcycle’ crops were extracted as the EOC’s test set, where EOC achieved an 11.06% error rate under 80% object identification confidence, tending to be less accurate than COC. Another three five-minute-long video clips were used as real-time video streams. Each node in the three ECs had one of the three clips.

The video query workflow after EOC’s deployment is shown in Figure 3. For each edge node receiving the real-time video stream from DG (Figure 3, Step 1), OD selected three consecutive frames with a specific interval (for example, 0.5s) and rapidly extracted crops potentially containing moving objects. Such crops were classified by EOC (Figure 3, Steps 2 and 3) and the results were used by IC for crop scheduling based on the Basic Policy (BP) (Figure 3, Steps 4 and 5). If the object identification confidence of a crop was above 80%, a targeted object was identified (predicted as positive due to the lack of ground truth of the real-time video), and its metadata was sent to RS (Figure 3, Steps 3, 6, and 7). If confidence was below 10%, the crop was dropped. Otherwise, the crop was sent to COC (Figure 3, Steps 3, 6, and 8). If the Top-5 classification results of the crop on COC contained the targeted label, a targeted object was identified (that is, predicted as positive) and its metadata was sent to RS (Figure 3, Steps 8 and 7). Since BP may induce queue backlog at EOC and frequent reprocessing at COC, we constructed an Advanced Policy (AP) (Figure 3, Steps 4 and 10) based on BP as a customized IC to further reduce E2E Inference Latency (EIL). AP collected and estimated EILs of EOC (Figure 3, Steps 5 and 4) and COC (Figure 3, Steps 9, 11, and 4) to guide crop uploading from OD (that is, load balancing, always sent to the one with a lower estimated EIL; Figure 3, Steps 2, 6, and 8). Then, AP reduced crops uploaded from EOC to COC by shrinking the identification confidence thresholds when either EOC’s or COC’s EIL got deteriorated.

**Application deployment.** As shown in Figure 4, we submitted a topology file to ACE, which was an extended YAML file containing meta information of both the application and all components. We illustrate the deployment of component OD as an instance. Receiving the topology file (Figure 4, Step 1), the orchestrator determined the node(s) (for example, Raspberry Pi ‘ec-1-rpi1’ on edge cloud ‘EC-1’) satisfying all requirements of OD—that is, ‘connections’ implying OD’s dependencies with components Local In-app Controller (LIC), EOC, and COC; ‘resources’ implying CPU and memory required by OD; and ‘labels’ implying that OD should be de-
ployed on edge nodes connected to cameras. Such decisions were recorded in the deployment plan ('instances'), a topology replica modified by the orchestrator. Note that, to manage nodes in an EC as a cluster, ACE can delegate node-level orchestration to the EC. Receiving the deployment plan, (Figure 4, Step 2), the controller transformed information of OD instances into specific deployment instructions in a standard Docker-compose YAML file, which was distributed to the node agent (for example, the container engine at ‘ec-1-rpi1’) for OD deployment.

Impact of implementation paradigm on intelligent application performance. We compared the performance of our application implemented with different paradigms. For CI, COC was deployed on CC. For EI, EOCs were deployed on ECs. For ECCI based on ACE, two versions of the application with BP (ACE) and AP (ACE+) were deployed. Different system loads were simulated by varying the sampling interval of frame differencing in OD from 0.5s to 0.1s. Since all comparatives used the same OD, we compared their video query performance using their object classification performance. Particularly, we used F1-score, edge-cloud BandWidth Consumption (BWC), and E2E Inference Latency (EIL) as evaluation metrics. We conducted the motorcycle query task under different system load and edge-cloud network delay (0ms for ideal and 50ms for practical) settings. Results are illustrated in Figure 5.

When the system load increases, F1-scores of CI and EI basically remain the same; CI using only COC and EI using only EOC achieve the highest and lowest F1-scores under all system loads, respectively. ACE and ACE+ using COC and EOC collaboratively manage to achieve F1-scores slightly lower than CI but significantly higher than EI. Unlike EI, in ACE and ACE+, many crops that cannot be confidently classified by EOCs (with a confidence from 10% to 80% and dropped by EI) are uploaded to COC. Compared with CI, few crops are dropped by EOCs (with a confidence below 10%) in ACE and ACE+. Besides, the higher the system load, the better ACE+ performs compared with ACE. Under higher system loads, more crops are directly uploaded from ODs to COC by IC with AP for load balancing in ACE+, reducing crops dropped by IC with BP in ACE. Furthermore, when the system load increases, ACE+ achieves higher F1-scores under practical than ideal network delay. In ACE+, under practical network delay, fewer crops are uploaded from EOCs to COC to avoid higher EILs by shrinking the confidence thresholds and more are from ODs to COC for load balancing.

When the system load increases,

Figure 5. Intelligent video query performance.
orchestration is challenging but critical to improving ACE-based application performance. ACE’s platform-layer orchestrator manages to allocate application components to proper nodes, satisfying basic (node-level) resource and user (edge/cloud deployment) requirements. However, fine-grained orchestration under more explicit constraints is still hard to achieve, which is significant to full infrastructure use. Furthermore, ACE’s static application orchestration cannot adjust to application or infrastructure changes. A dynamic orchestrator is also necessary.

Resource-contention prevention must be investigated further to ensure the performance of ECCI applications co-located at the same infrastructure. Currently, ACE achieves component-level resource isolation through containerization and supports inter-component resource allocation optimization through the customized in-app controller, where, however, application-level resource isolation and allocation is still an open issue. Critical resources such as edge-cloud bandwidth should be allocated appropriately to co-located applications under ACE’s coordination. It is also promising to integrate the serverless technology for elastic resource allocation that cannot be directly achieved by container-based solutions.

Security is another critical issue. ACE now contains no security module, where state-of-the-art encryption and authentication techniques can be directly integrated for fundamental secrecy. The actual challenge, however, is access control. In our design, ACE users have full access to their infrastructure and ECCI applications, where no user collaboration is currently supported. For specific applications (for example, federated learning) that must be developed and deployed collaboratively by multiple users, ACE is required to provide a fine-grained access-control mechanism. It must ensure each collaborator has limited access to the shared application and infrastructure without jeopardizing others’ privacy.

Future of ACE
As a prototype for cost-efficient ECCI application development and deployment, ACE is still in its infancy. The construction of ACE reveals fundamental challenges which must be addressed and sheds light on the vision of an ACE-based ECCI ecosystem deserving exploration.

Challenges. Agile ECCI application orchestration is challenging but critical to improving ACE-based application performance. ACE’s platform-layer orchestrator manages to allocate application components to proper nodes, satisfying basic (node-level) resource and user (edge/cloud deployment) requirements. However, fine-grained orchestration under more explicit constraints is still hard to achieve, which is significant to full infrastructure use. Furthermore, ACE’s static application orchestration cannot adjust to application or infrastructure changes. A dynamic orchestrator is also necessary.

Resource-contention prevention must be investigated further to ensure the performance of ECCI applications co-located at the same infrastructure. Currently, ACE achieves component-level resource isolation through containerization and supports inter-component resource allocation optimization through the customized in-app controller, where, however, application-level resource isolation and allocation is still an open issue. Critical resources such as edge-cloud bandwidth should be allocated appropriately to co-located applications under ACE’s coordination. It is also promising to integrate the serverless technology for elastic resource allocation that cannot be directly achieved by container-based solutions.

Security is another critical issue. ACE now contains no security module, where state-of-the-art encryption and authentication techniques can be directly integrated for fundamental secrecy. The actual challenge, however, is access control. In our design, ACE users have full access to their infrastructure and ECCI applications, where no user collaboration is currently supported. For specific applications (for example, federated learning) that must be developed and deployed collaboratively by multiple users, ACE is required to provide a fine-grained access-control mechanism. It must ensure each collaborator has limited access to the shared application and infrastructure without jeopardizing others’ privacy.

Vision. ACE demonstrates the potential in supporting closed-loop DevOps of ECCI applications. ACE facilitates the cost-efficient, effective development
and deployment of ECCI applications. Taking a step further, we believe it is viable to integrate proper operation and maintenance modules into ACE, aiming at the closed loop of continuous ECCI application development, deployment, monitoring, delivery, and testing. Such complete DevOps support will enable ACE to act as the foundation of the approaching ECCI ecosystem.

ACE promises to promote a broad spectrum of production-level ECCI applications. ECCI applications, especially high-performing ones, are difficult to design, develop, and deploy, which hinders such a paradigm from contributing to the rapidly expanding IA market. ACE supports the entire ECCI application lifecycle, helping general users to conduct unified and user-friendly application development and deployment. Besides, ACE can also ease the migration of existing IAs based on CI and EI to ECCI applications, satisfying specific practical requirements.

Conclusion
ML/DL-based IAs with harsher practical requirements cast challenges on conventional CI implementations. The emerging ECCI paradigm can support proliferating IAs that are, however, currently developed and deployed individually without generality. We envision systematic designs of a unified platform for cost-efficient development and deployment of high-performing ECCI applications, guiding us to construct the ACE platform handling heterogeneous infrastructures and IA workloads. Our initial experience shows that ACE helps developers and operators along the entire lifecycle of ECCI applications, where customizable optimizations can be efficiently conducted. More research is still required, and we have discussed both the challenges and visions of the newborn ACE.

Acknowledgments
This work was supported in part by the National Key Research and Development Program of China under Grant 2020YFA0131900; the National Natural Science Foundation of China under Grants 61772410, 61802298, 62172329, U1811461, U21A6005, 11690011; the China Postdoctoral Science Foundation under Grants 2020T130513, 2019M663672; and the Alan Turing Institute.

References
1. Abdelazeh, T.F. et al. Five challenges in cloud-enabled intelligence and control. ACM Transactions on Internet Technology 20(1), 2020, 33–319.
2. Alvar, S.R. and Bajic, I.V. Pareto-optimal bit allocation for collaborative intelligence. IEEE Transactions on Image Processing 30 (2021), 3348–3361.
3. Apple. Private federated learning. NeurIPS 2019; http://bit.ly/3GgLOp.
4. Bagchi, S., Siddiqui, M.B., Wood, P., and Zhang, H. Dependability and correctness of distributed algorithms for the ACM 63, 1 (2019), 58–66.
5. Bianchini, R. et al. Toward ML-centric cloud platforms. Communications of the ACM 63, 2 (2020), 50–59.
6. Castro, P., Ihschian, Y., Multiusany, V., and Slominski, A. The rise of serverless computing. Communications of the ACM 62, 12 (2019), 44–54.
7. Chai, H. and Bajic, I.V. Deep feature compression for collaborative object detection. In 2018 25th IEEE Intern. Conf. on Image Processing, 3743–3747.
8. Chung, J.-W., Kim, J.-T., and Moon, S.-M. ShadowTutor: Distributed partial distillation for mobile video AI inference. 49th Intern. Conf. on Parallel Processing (August 2020), 81–811.
9. Eshratifar, A.E., Abrishami, M.S., and Pedram, M. JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services. IEEE Transactions on Mobile Computing 20 (2), 2021, 568–582.
10. Gong, Y. et al. Edgetorch: A recommender system on Edge in mobile Taobao. 29th ACM Intern. Conf. on Information and Knowledge Management (October 2020), 2477–2486.
11. McMahan, B. and Ramage, D. Federated learning collaborative. Google Research (April 6, 2017); http://bit.ly/3E9eOW8.
12. Doucet G. and Gassier, E. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. European Conf. on Information Retrieval (2020); Advances in Information Retrieval. Springer, 345–359.
13. Greenwood, S. AI on Edge. Communications of the ACM 63, 9 (2020), 18–20.
14. Harchol, Y. et al. Making edge-computing resilient. 2019 Master’s thesis, EECS Dept., University of California, Berkeley.
15. He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. 2016 IEEE Conf. on Computer Vision and Pattern Recognition. T70–T78.
16. Hong S. et al. ExaME: a big data on-chip accelerator. Ensemble serving for deep learning models in intensive care units. 29th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining. (August 2020), 1814–1824.
17. Huang, J. et al. Enabling automatic compilation of deep learning pipelines across IoT and cloud. In Proceedings of the 20th Annual Intern. Conf. on Mobile Computing and Multimedia (April 2020), 581–5812.
18. Hung, C. et al. VideoEdge: Processing camera streams using hierarchical clusters. 2018 IEEE/ACM Symp. on Edge Computing, 115–131.
19. Kang, Y. et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In Proceedings of the 22nd Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (April 2017), 615–629.
20. Kim, S.W. et al. Learning to simulate dynamic environments with GameGAN. 2020 IEEE/IFCV Conf. on Computer Vision and Pattern Recognition. 12238–12247.
21. Krizhevsky, A., Sutskever, I., and Hinton, G.E. ImageNet classification with deep convolutional neural networks. Communications of the ACM 60, 6 (2017), 84–90.
22. Laskaridis, S. et al. SPIN: Synergistic progressive inference of neural networks over device and cloud. 29th Annual Intern. Conf. on Mobile Computing and Networking (2020), 371–375.
23. Li, S. and Lan, T. HotDedupe: Managing hot data storage at network edge through optimal distributed deduplication. IEEE Conf. on Computer Communications (2020), 247–256.
24. Li, Y. et al. Reducto: On-chip monitoring for resource-efficient real-time video analytics. In Proceedings of the ACM SIGCOMM (July 2020), 358–376.
25. Lipton, P. Mosquito man; https://mosquito.org/man/mosquito-B.html.