Dash: A Low Code Development Platform for AI Applications in Industry

Yifan Wang
Department of Computer Science
Cornell University
Ithaca, New York, USA
yw2399@cornell.edu

Weijia Song
Department of Computer Science
Cornell University
Ithaca, New York, USA
ws393@cornell.edu

Yuting Yang
Department of Computer Science
Cornell University
Ithaca, New York, USA
yy354@cornell.edu

Charif Mahmoudi
Siemens Corporation, Technology
Princeton, New Jersey, USA
charif.mahmoudi@siemens.com

Shashank Shekhar
Siemens Corporation, Technology
Princeton, New Jersey, USA
shashankshekhar@siemens.com

Kenneth P. Birman
Department of Computer Science
Cornell University
Ithaca, New York, USA
ken@cs.cornell.edu

***Abstract*—Low Code Development (LCD) is a popular way of creating database applications owing to its simplicity and the resulting improvement in accessibility: the developer uses gestures like drag and drop with icons representing existing programs or data sources, then employs pull-down menus for customization, and hence rarely needs to write new software. Our work is motivated by a recent trend that seeks to move the LCD methodology into AI settings, such as for product inspection in industrial shop floors (factories). We start by showing that LCD-created AIs (LCDAIs) for time-pressured tasks often need so much customization that the deployment engineer ends up needing a deep understanding of the application, subverting the LCD philosophy. Dash introduces a new LCD platform that focuses on LCDAIs for complex, time-sensitive distributed systems. Beyond these basic goals, the technology optimizes both development and deployment.**

***Index Terms*—low-code programming, cloud computing, edge computing, artificial intelligence**

I. Introduction

Low-code development has obvious appeal for many end-users. Rather than viewing coding as a prerequisite for application development, LCD systems treat software creation as a task centered on the creation of a data-flow logic diagram. Developers drag, drop, and connect provided modules, or customize generic ones using pull-down menus. Some represent input or output; others correspond to standardized computational actions.

LCDAIs bring artificial intelligence into what traditionally would have been a data-centric task. In a typical use, an LCDAI might capture imagery and then leverage ML for image segmentation, identification of objects, scrutiny of the identified objects, and so forth. This lets the deployment specialist focus on the core logic of the application without needing to create new AIs, dive deeply into data science or machine learning, or perform other tasks that are typically viewed as part of the coding process. The demand for such solutions is strong, and software vendors are releasing a wave of LCD platforms, such as three we will look at more closely: Nvidia Composer [1], Azure ML Designer [2] and AWS SageMaker Canvas [3].

LCDAI creation generally involves two stages. First, an AI expert (using LCD platform tools but also doing a fair amount of hands-on parameter tuning and perhaps some actual coding) creates an AI application as a data-flow graph. The LCD platform is helpful, but substantial expertise is still required: this AI expert will be an expensive, highly trained individual. The resulting LCDAIs are not, however, ready for instant deployment. Instead, they should be viewed as *templates* that will often need further customization at deployment time. A stage in such a template, i.e., a node in the data-flow graph, designates the AI programs to run, the names for file outputs and inputs; edges represent data flow.

Accordingly, when it is time to install the LCDAI and configure it for a specific use case, more work will be required: in effect, the template needs to be instantiated. We consider this to be the job of a deployment specialist, who must fine-tune the LCDAI to create a customized version optimized for use in a specific setting (such as a parts inspection station on a shop floor). Customization may involve designating specific cameras, selecting the best AI model for the general-purpose AI components in the application, fine-tuning to deal with specific lighting or conveyor-belt speeds or other, etc.

To maximize the return on investment, any company selling LCDAI solutions will want to sell many instances of each of its LCDAI templates, and hence aims to hire a small number of AI experts, but then to sell large numbers of instances. Because we are quite far from a science of self-configuring and hence full-automated LCDAI deployment, the ratio of deployment specialists to AI experts would be high. The vendor will instead aim for a semi-automated LCDAI template that guides the deployment expert through the needed tasks, in the hope of minimizing the frequency of *escalation*, namely tasks for which the AI expert themselves need to become directly involved.

Our work started with a review of today's commercial

options. We determined that LCDAI creation falls short in this last respect by requiring too much back-and-forth in the two-step development: too many issues require hands-on attention from the AI expert. This leads to the three areas in which our new system, Dash, innovates:

1) Dash provides better AI development support for the AI expert, particularly in the area of AI model selection.
2) Dash recognizes that the AI deployment specialist is not a coder or AI expert, and provides a friendlier deployment experience that includes scripted support for fine-tuning an AI (for example by training on example images that accurately reflect lighting or other local conditions) without forcing the deployment specialist to deeply understand how ML training is implemented.
3) Dash provides better runtime support, enabling field diagnosis and correction of problems that might currently occur to the AI expert.

The remainder of our paper is organized as follows. Section 2 amplifies on these points, illustrating them in the context of a product inspection station for a factory (a "shop floor" scenario, to use standard terminology). This leads to a restatement of the challenges confronting the current generation of LCD platforms when applied to LCDAI creation. Section 3 reviews three popular LCD platforms relative to these goals and challenges. In Section 4, we introduce the architecture and features of Dash followed by a proof-of-concept evaluation in Section 5. Section 6 summarizes our plans for future work, and Section 7 concludes the paper.

II. A Product Inspection Station

To illustrate the challenges, consider the creation of a defect detector using photos taken from a conveyor belt carrying components from a Computer Numerical Control (CNC) fabrication unit. Often, the CNC machine takes input materials like sheets of metal or plastic and uses a cut-and-stamp process to create parts that will become input to later processes. To minimize waste, a single piece of metal or plastic will often yield many parts, and these parts may be of different types. Thus the task arises of sorting the parts and assessing each for quality.

The LCDAI task we considered arises when a batch of parts in different shapes arrives at an inspection station on a belt, with each part potentially having its own distinct part-specification. The defect detector needs to first recognize parts by their shapes, then route each image to a specialized AI (or AI pipeline) designed to verify that the part is properly fabricated. For example, if the part has holes drilled in it, the LCDAI first checks that the part as a whole was properly formed, then checks that the holes are in the proper locations, and then checks for clinging chaff or cracks. These tasks are most effectively performed using different machine vision techniques, hence we will often expect that each check should be done by a separate AI (running as a separate process). In effect, the LCDAI will take the form of a distributed pipeline or graph, with the first step determining part shape, and subsequent steps doing shape-specific tasks. The distributed nature of this LCDAI lends itself to improved performance from parallelism and because each subtask is performed by a different stage, can even draw on different ML packages: perhaps, TensorFlow [4] for one, PyTorch [5] for another, etc. Thus an LCDAI will often be a distributed program, and will need to be mapped to suitable hosts. The AI expert will create a flexible template able to accommodate a variety of configurations, but even the number of part inspection pipelines will depend on the details of the use scenario.

Accordingly, the template will include meta-logic, such as "foreach" loop that in fact loops over information obtained during deployment: when the deployment specialist is given the specification of the CNC output, we learn how many distinct parts are being created, how each looks, and which quality checks to perform.

The AI expert would start by selecting or creating a general-purpose AI for classifying parts: given an image of a bin holding a mixture of parts, this entails using image recognition to figure out which part is which (there might even be a prior step of pushing the parts around to ensure that they are lying flat in a single layer, rather then being piled one on another).

Next, for each part that was identified, the LCDAI would route the image to the corresponding quality assurance and defect analysis pipelines: the LCDAI would be a graph with one of these pipelines per category of parts.

From this example, it should be clear that the template created by AI experts leaves many unfinished LCDAI "configuration" tasks. These are intended to be performed by the LCDAI deployment specialist: a person who understands the shop floor domain, but in principle is not trained in coding and related tasks. Thus it is only during deployment that inspection-station-specific considerations enter the picture. For example, the actual parts that might be examined at one station using this template could be different from at some other station (perhaps one is fabricating small stamped metal parts and another, larger molded plastic parts). Vision tools often must be fine-tuned to deal with color contrasts, lighting, possible object orientation, the possibility of partial occlusion (one part that happens to land on top of another), etc. Indeed, many cameras need to be configured to tell the camera itself what spectral settings to use (some flaws are much more visible in UV light, or infrared light, or with polarization filters). Some devices maybe used only as needed: an ultrasound or X-ray unit, for example. The template will thus contain general purpose AI modules that need to be fine-tuned by the deployment specialist who will participate in setting up the inspection station on the floor shop.

The overall approach thus poses three kinds of challenges.

- During the AI creation stage, the AI expert will be faced with a lack of "compositional" off-the-shelf tools. Knowing the best-of-breed solutions for each subtask doesn't imply that template creation will simply entail snapping the stages together like lego parts. A task like image segmentation (taking a photo and finding objects within it) might not be able to send its output to a follow-on task like object identification without work to ensure

compatibility between earlier AI stage and the follow-in stage, in the sense of data being passes from one to the other. This entails making choices about image sizes, resolution, file formats, spectral data, and lighting assumptions. It is not uncommon to introduce scripting or coding to transform from one format to another, for example to change the resolution of an image. Work is also needed when parameterizing the LCD pipeline: each step must run fast enough for a given task and yet also must achieve required accuracy. Like any program, a templated LCDAI would need to be tested, and this entails mocking up some inspection stations, creating various scenarios, deploying the LCDAI and making sure it runs fast enough and achieves adequate accuracy. All of these actions require deep insight into both the LCD candidate and the application.

- No two factories are created the same, and the customizations needed during deployment may involve a wide range of "runtime" considerations reflecting physical conditions on the shop-floor. In today's approach to AI creation, these physical constraints imply that the AI expert must assume a hands-on role in deployment, looking for an existing model with the proper characteristics and creating a new one if needed. Our objective is to treat customization as part of a two-stage activity in which the second stage can be performed by a deployment specialist who wouldn't be trained in the AI technologies the AI expert will have selected, and wouldn't be able to create code fragments – even simple scripts that simply resize a photo. Lacking specific attention to such matters, each deployment will encounter a dozen (or even dozens) of such customization needs, and each time the deployment specialist will end up escalating and waiting for help from the AI expert.
- Mistakes and surprises are inevitable in any software undertaking. There will often be a need to troubleshoot or even "debug" the customized and deployed LCDAIs in the actual factory. Again, we want to view this as a routine part of the undertaking, and hence we would want to do so without breaking the LCD model. In our two-step methodology it is the deployment specialist and not the AI expert who must "installs" (instantiates) the template, and would become aware if the resulting LCDAI is not sufficiently fast, sufficiently accurate, etc. While major issues should escalate to the AI expert, minor configuration errors should be resolvable without requiring expert help.

Our project, Dash, aims at all three challenges described above. This was a large set however, and we see Dash as a multiyear undertaking that will yield a large prototype. The present focuses mostly on the experience of the AI expert, with follow-on papers planned that will do deep dives on other aspects of the challenges. Dash itself is intended to be a complete system: a fully realized research prototype suitable for deployment and experimental use in real settings, but not a product that we would sell: the hope is to learn to create such a product, but to focus on fundamentals and leave "polish" such as fancy GUIs for future commercial follow-ons.

Accordingly, this paper focuses on Dash's roles during AI creation and how the AI expert can anticipate and facilitate the deployment process, emphasizing mechanisms that free the deployment specialist from the need to do extensive coding or needing to deeply understand how the LCDAI works. At the same time, Dash is intended to be a component of a software ecosystem. Specifically, our work assumes that a shop floor would typically have a cluster of standard compute servers on which data and computation mostly occurs, configured as an AI hub. Product inspection stations and other sensors or actuators (such as robot arms with their own robotic control units) would connect to the hub over standard fast, low-latency, networking technology. The assumption that the edge system is a reasonably powerful cluster allows us to assume that the LCDAI applications Dash generates will run in a standard "scalable edge computing" execution environment that can also include file systems, database systems, standard connectors such as Kafka [6], cluster management tools like Kubernetes [7] or Apache YARN [8], or entire data analytic stacks such as the Apache framework. On the other hand, the edge compute cluster will not have the incredible compute power of a full-blown cloud: it might have 2 or 4 servers, whereas a cloud could have hundreds of thousands.

The Dash prototype works particularly well over Cascade, Cornell's new data backplane and compute-hosting platform [9]–[11]. Cascade was created to host time-sensitive AI tasks on edge clusters of the sort just described, hence given sufficiently performant AI and adequate low-latency bandwidth, Dash on Cascade can offer unified development and deployment environment appropriate for the shop-floor setting.

What timing constraints are seen in our defect detection example? Clearly, inspection needs to be completed while the product is still within the inspection station, and if a malformed part should be removed from the conveyor belt the robotic arm that will take that action needs to be in motion soon enough to grab the part while it is within reach. Even this example highlights a tradeoff: higher-resolution images are larger, and hence will take more time to transmit, hence potentially slowing the peak speed of the conveyor belt. The AI expert will ultimately need to "control" those choices. Later, though, at runtime, one can image a deployment that fails to satisfy implicit assumption the AI expert made: perhaps, they developed and tested on dedicated hardware but the compute servers used in the deployed setting are actually shared, or has a slower network or CPU, or there is some other relevant difference. Some problems should simply be reported to the deployment engineer; some may need work to fine-tune the LCDAI in a way anticipated by the AI expert, and a few might require escalation to that AI expert. Dash does not address all such cases today, but our vision and research plan includes tackling these and trying to automate as much as feasible, using an LCDAI mindset throughout.

III. Popular LCDAI Platform Review

Before assuming that a new kind of LCDAI platform is needed, we reviewed popular LCDAI products relative to our target setting: Microsoft Azure ML Designer, AWS SageMaker Canvas and Nvidia Composer[1]. We consider three perspectives: runtime environment, image data support, and AI programming.

A. Application Service Environment

A significant differentiator between these products involves the servers on which the generated application runs. Designer and Canvas are cloud-based solutions, integrated deeply with the respective cloud infrastructures. Composer, in contrast, targets a containerized deployment that might run on a machine external to the cloud.

Applications developed using Designer and Canvas are first built in the LCD style outlined earlier, then deployed by moving virtual machines into the cloud environment: Azure or AWS respectively. The cloud makes deployment easy: users only need to specify the target virtual machine name and with a few mouse clicks, the application can be up and running. However, this simplicity comes at a cost: high latency. For our running defect detection example, in a typical shop floor setting the image analysis stage must complete within a few tens or hundreds of milliseconds to ensure that the robot arm positioned a meter down the conveyor belt will have time to position itself to pick up defective parts before the part moves further down the conveyor towards the next stage in the manufacturing process. Designer and Canvas offer no explicit modelling of time and make extensive use of cloud services that themselves lack real-time functionality. The AI expert lacks the "hooks" that would be needed to describe deadlines or priorities, and lacks ways to ensure that deadlines will be respected.

Composer is less specific to cloud environments: It outputs containers that can be managed by schedulers such as Kubernetes or Apache YARN. This decision, however, brings a different set of limitations: Composer must assume less about the runtime environment than Designer and Canvas are able to, and hence the deployment engineer will need hands-on involvement if the LCDAI will interact with services of the kinds we find in general clouds. The consequence is evident when we think about easy of deployment: to install this sort of LCDAI, the deployment specialist would often need a deep understanding of the logic of the LCDAI in order to wire it to the appropriate local files, databases, and other services.

B. Support for Large Image and Video Data Objects

Factory AI applications require large inputs (videos, photos, ultrasound, etc) [12]. Typically one attaches the selected imaging devices to a dedicated hosting computer running an RTOS, with dedicated high-speed connectors and a specialized camera-control application written just for the purpose. These RTOS hosting units are limited to managing the device, doing basic filtering tasks to discard unwanted images, and uploading "interesting" ones. Heavier tasks such as the AI ones just listed will thus run on a nearby cluster hosting the LCDAI pipeline, accessible over a wired optical network with gigabit bandwidths and low latencies. One can see why this should work, but none of the three LCD AI products we examined assumes this deployment style. Thus the deployment expert would be involved at every step: establishing connectivity, deciding how the LCDAI will be mapped to cluster hosts, GPUs and FPGAs, loading any needed GPU kernel libraries, etc.

The cloud-hosting of Designer and Canvas rule these products out for many use cases. Cloud data centers are generally sited in settings with inexpensive power and cooling [13], but factories are situated in settings selected for product-specific reasons such as convenient raw materials or shipping, local employee skill level [14], etc. As a result they may be physically far from one-another, and this can preclude ultra-low latency, high-bandwidth links to the cloud. If all images must first be sent to a data center that is hundreds of miles away, the game may be over before the image even reaches Azure or AWS. Moreover, today's clouds favor a store-then-analyze model in which images are first stored into a repository and the AI is triggered only after storage completes. Designer's storage is based on Azure's big-data storage infrastructure, such as Cosmos DB for structured (tabular) data, the Binary Large Objects (BLOB) store for datasets like photos and video, and the Azure Data Lake for huge collections of semi-structured and unstructured content. This design inevitably incurs substantial latency. Thus Designer is biased towards maximizing throughput using pipelines and batching. While Designer does support a real-time mode, that mode has limited ability to base decisions on a changing context – indeed, it can be recognized as an instance of the classic cloud model in which scalable microservices respond to web queries with low latency, but updates are streamed to backend systems that might not push new versions of indices and models until after a substantial delay. Some factories may be lucky enough to be near a suitable cloud, and some shop floor applications could work within this model, but there are many where it would be impractical.

In our view, Designer's batch mode is focused on what would classically be the update pipeline in a cloud used to provision web applications and web sites. In such a model asynchronous events are collected and uploaded as a set – a batch. To enable batch mode, Designer would be configured to buffer data, for example in Kafka or in the Blob Storage. When the batch fills, or if a timeout expires, the entire set of requests is then processed. Batching is highly beneficial from a throughput and bandwidth perspective, but simply isn't "about" minimizing latency, hence successful deployment in a time-sensitive setting (at least one with deadlines in the hundreds of milliseconds range) is very much a matter of luck.

In contrast to Designer and Canvas, Composer does not

[1]Notice that we are not considering graphical AI builders such as the Spark or Apache application editing GUIs because both focus on assisting code creators, rather than LCD development. Similarly, we do not consider database visualization tools because those focus on data analysis, not AI actions.

provide built-in data backbone, nor does it provide any data transmission support. Deployment specialists need to tune the container cluster to run the Composer-generated applications successfully and to meet any specific performance needs.

C. Support for AI

Given that these three products are all intended as LCD development solutions for AI, what kinds of explicit AI support to they offer? All three are compatible with libraries of prebuilt AI models intended to meet the users' general needs. They then encourage users to fine-tune the general-purpose models for their special down-stream tasks.

For instance, these platforms all support YOLO [15] in different versions as a base model for object detection, which can then be fine-tuned by the application developer (as in our defection detector example, to identify parts by categorizing their shapes, or to enable quality controls by locating drilled holes). In addition to the built-in models, vendors also support public libraries, e.g., Transformers [16], and hence users can include the popular large models like ViT [17] in their work. Here, though, we encounter an interesting new issue: the lack of tools to help the users select the most suitable AI model when a set of models are all potential candidates, and a lack of tools to facilitate the debugging process if a problem arises.

Furthermore, none of the platforms acknowledges the two-stage approach we outlined in the introduction, where an AI expert creates a template that would later be specialized by a team of deployment experts who need to customize the solution without re-involving the AI expert who created the template. In effect, they assume that the same AI expert who designed the template will also be responsible for deployment and any needed customization or tuning.

Indeed, thinking about this two-stage approach we find that even with respect to the AI expert, existing LCD platforms fall short. Let's drill down on this question of picking the ideal AI. Even if we hold the AI technique and codebase constant, an AI also requires hyperparameters and a trained model, and may depend on additional configuration actions. Model selection (when a system has a set of available models all pretrained, but differing by having been trained on different data sets) is an often-overlooked aspect of AI deployment, but highly relevant in our target settings.

Consider our running defect-detection example. It does start with image segmentation and object classification, but then treats different objects in different ways: objects of type A will be checked for defects by the A-component quality assurance pipeline, objects of type B by the B-component pipeline, etc. Moreover, many images might be captured but then discarded immediately as "uninteresting". For objects worth examining, there are a whole series of configuration choices that must be made: input format, resolution, etc. The target environment, which may have its own lighting characteristics, shadows, color choices, etc. Thus our setting seems to require hyperparameter optimization and model training *at deployment time*. But in our LCDAI goals, the deployment specialist is assumed to lack the requisite training and coding ability, and would not be able to perform this task.

Yet over time, an LCDAI vendor might quickly build up a large "library" of hyperparameter and model pairs. The lighting and actions at this inspection may have been encountered at dozens or hundreds of other inspection stations in the factor. Even the collection of parts being examined might be common to a whole series of fabrication units. Thus, before we escalate to the AI expert, the LCDAI deployment "system" really should check to see if the needed model already exists! Yet this task could be subtle. It is not as simple as to say "pick a computer vision model for UV lighting with a polarizing filter". Very likely we would need the deployment specialist to capture sample images, then run through a set of candidate models to see if any of the "off the shelf" options is suitable, and we would only escalate to the AI expert in a genuinely new situation. In fact the AI expert, *anticipating* this entire situation, may want to create a script that would walk the deployment specialist through the task. With such a script, the AI expert could even design a case-specific quality metric so that if there are a half dozen options that match in a superficial sense, the LCDAI deployment will still opt for the very best choice.

The model selection question arises when we hold the choice of software (algorithm and implementation) constant, but in many cases there are higher level choices to be made, too. YOLO, as an example, has different versions that are optimized towards different goals. For our sample task of defect-detection, deployment specialists might choose YOLOv7 [18] as the object detection model if accuracy is the main concern, or they might choose YOLOv5 [19] instead to achieve a shorter inference time. Thus there are multiple levels of choices to be made. Unfortunately, none of the platforms we reviewed offers effective tools to help users select among a set of candidate models when this form of choice arises. Once a developer makes a tentative choice there are no tools to help verify the efficacy of their choice, such as its ability to meet timing goals, its likely accuracy, etc. Designer does include a catalog of example applications for the users to build upon and customize but lacks any form of automated model selection support.

Debugging exposes further needs. The major debugging method in all three platforms is logging: the developer is expected to run the application. The framework offers a centralized logging feature, and the AI expert is supposed to analyze the log from a problematic run to identify and interpret errors that might have been reported. Here, Designer goes a bit further: it includes a tool that will compare the log from a failed run with a log from a successful run to help the user understand what went wrong. But this form of "diff" would not extend to subtle timing problems (for example, if stage 2 of a 3-stage pipeline was sometimes slow and this can cause stage 3 to exceed a deadline, Designer would not point to stage 2 as the likely culprit).

The details of the 3 platforms are summarized in Table 1. Notice that we have also included Dash, although this work

TABLE I
LCDAI PLATFORM FEATURE SUMMARY

Popular LCD Platforms		Azure ML Designer	AWS SageMaker Canvas	Nvidia Composer	Dash
Service Environment		Cloud	Cloud	Local/Cloud	Local/Cloud
Data Support	Batch Mode	✓	✓	✗	✓
	Real-time Mode	✓	✓	✗	✓
	Data Transmission Protocol	HTTPS	HTTPS	✗	HTTPS / RDMA
	Data Backbone	Cosmos DB, Blob Storage	Dynamo DB, S3	✗	Cascade, Kafka
AI Support	Model Selection	App Example	✗	✗	Hyperparameter-based recommendation
	Debug	Log Application Comparison	Log	✗	Log Type Checking

is actually discussed in the next section.

IV. FEATURES AND IMPLEMENTATION

We now describe Dash, our experimental platform for creating LCDAI solutions. Dash seeks to address all the challenges discussed above. As noted earlier, our work to date has focused on the AI expert. Dash features that focus more on template instantiation by the deployment engineer and on runtime monitoring and debugging will be discussed in a future work.

A. Dash is the first LCD platform built on performant distributed data backbone.

Our work on Dash revolves around a premise that seems evident, yet differentiates the system strongly from most other projects. Whereas existing LCD platforms are often designed to emit VMs, containers, or some form of database triggers that will run when a trigger condition is satisfied, Dash treats the LCD target as a distributed program that will be deployed into a distributed edge infrastructure. This helps explain why we focus on Cascade – a compute and storage hosting framework designed for high speed edge systems – as our preferred Dash infrastructure (we also support the Apache software stack). When running on Cascade, all data flow logic (including the AI models, data processing, conditional control, iteration) is represented in a data flow graph (DFG). Nodes in the DFG correspond to user-supplied AI that would typically run in a dynamically loaded module (a DLL). Edges represent information sharing between AI modules through files, pub/sub notifications or key-value puts – Cascade supports all three models (mapping the first two into key-value operations and treating the KVS API as its most fundamental one). Dash, then, has the role of constructing a new DFG and assisting the deployment expert by properly registering the AI expert's selected UDLs within Cascade so that when a triggering event occurs, Cascade can schedule the DFG within the cluster in a manner designed to prioritize real-time responsiveness as a primary goal, but to maximize utilization when doing so will not compromise latency.

Both Dash LCDAI templates and LCDAI deployments use the same DFG model; the primary difference is that for a template, the hyperparameters, model parameters and additional inputs to the UFG nodes are considered to be "initial" versions but perhaps not the final versions that will be used in the deployed system. A semi-automated instantiation procedure, defined by the AI expert, is used to guide the deployment engineer through the more detailed setup and testing that must occur when the template is brought into a concrete environment. This work is still in progress, but our plan is that node by node, Dash would assist the deployment engineer in connecting the AI component to its inputs (for example by dragging the proper camera onto a "required input" node in the DFG), configuring the camera (for example by sending a JSON configuration file to the camera-hosting program), and then testing the setup (for example by capturing test images with known content). The script could then self-evaluate to determine whether the UDL is performing adequately, and if not, direct the deployment engineer to run a few-shot training procedure or to make other adjustments to the deployment scenario.

To maximize compatibility with the Apache software stack, Dash treats publish-subscribe as its primary tool for implementing edges between UDL nodes in the LCDAI DFG. When an event will trigger a pipeline of AIs – a sequence of UDLs – Dash publishes the event as an input to the first UDL, and then assists by configuring that UDL to publish its output as input to the next UDL in the DFG, etc. This chain of publication-subscription goes on to form the entire dataflow logics, as represented in the grey area of Dash Data Flow in Fig. 1.

Notice that in the figure, Dash is shown as being "outside" the Cascade hosted service, and also distinct from network management and edge client infrastructure (the edge clients are the nodes hosting input devices). Dash per-se is best understood as a kind of GUI, used in an offline role: active during AI design, and active during deployment, but staying out of the way during production. Down the road we do expect to add a variety of on-the-fly monitoring options aimed at assessing LCDAI performance, but we are wary of slowing down latency-focused LCDAIs in order to instrument them. In our initial implementation, Dash has no runtime overheads of any kind.

The Cascade publish-subscribe layer maps to a KVS implementation, but looks and behaves exactly like Kafka, the widely popular publish-subscribe tool. This enables Dash to interoperate seamlessly with Kafka-based applications, and also enables Dash to benefit from high quality network management. Because Dash is not on the runtime critical path,

the LCDAI can benefit from the full performance of the factory network. Cascade supports local deployment using both RDMA and TCP as options for data transmission; the former is capable of fully loading a 100Gb (12.5 GB) optical network and adequate even for transmission of large, high-resolution files. Because Dash is aware of deadlines, data formats and throughout requirements, it can export this kind of information as JSON files suitable for use as inputs to network monitoring tools, network management tools, and other frameworks.

B. Integration of model composition with type checking.

The process of selecting the correct models and picking their parameters (including both hyperparameters and configurable parameters) and to populate templates is time-consuming and error prone, and in some situations the needed model may not exist: the AI expert might need to step back in and train a new model using data from the runtime setting. Dash seeks to ease this step by recommending the closest fitting models or a most suitable set of parameters. Dash also considers feasible ways for deployment engineers to run experiments, evaluating the effectiveness of the solution, and perhaps reverting to the application developer for help retraining certain components.

To recommend the most closely fitting model, Dash uses a weighted distance to determine the similarities between training environments and real environments. The equation is as follows.

$$d = \sum_{k=0}^{n} w_k (r_k – t_k)^2 \tag{1}$$

In the equation, n is the number of parameters. r_k is the real parameter value measured from the shop floor, e.g., the ambient light has a color temperature of 5000K, as a common factory lighting condition. t_k is the expected parameter value from the model, e.g., a vision model might be trained with pictures taken from outdoor settings where the color temperature is 7000K. w_k is the non-negative weight designated by the AI experts. The bigger the weight, the more important a parameter is. If the AI experts believe color temperature should not impact the model's performance, they can set the value to 0. So, the smaller the distance is, the closer a working environment is to the model. In this way, Dash can recommend models from the ensemble provided by the AI experts.

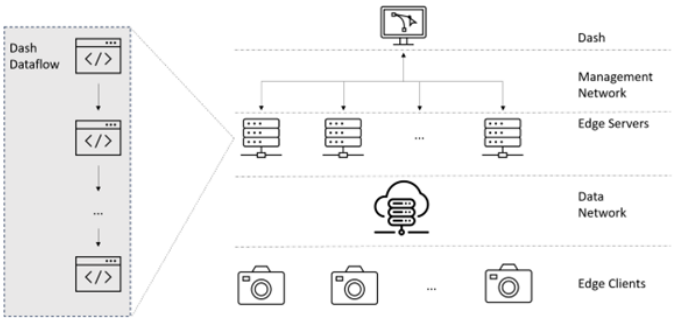

Fig. 1. Dash Architecture.

We recognize that the weighted distance algorithm is not perfect. The first issue is that it cannot handle the scales well. The rparameters come in vastly different scales, like the color temperature is at the magnitude of thousand, while the learning rate is tuned at the step size of thousandth. Widely accepted mathematical transformation methods like unification don't work well either since parameters might not have a clear range or the range is too broad. Another drawback is that a simple addition of the squared distance hides the physical meaning behind the parameters. For computer vision models, it doesn't make sense to add the difference in color temperature to the difference in image resolution. To solve the issue and offer a general solution, Dash allows AI experts to provide their model picking methods. We'll discuss the model picking mechanism in future work too.

In addition to the model recommendation feature, Dash also helps debug the AI application by offering type checking mechanism at program module level. When there is a change made in the data flow, Dash checks and makes sure that the changed module type checks with its previous and next module in the pipeline. To perform the type checking, Dash checks both the primitive data type and the data matrix dimension. Type checking can effectively reduce the number of runtime errors that the deployment specialists might encounter.

C. Dash eases deployment, especially in complex settings.

There are 2 major sources of complexity that slow down the pace of application creation and deployment. The first is a failure to acknowledge and embrace the huge differences between development and deployment of LCD solutions, while the second stems from heterogeneity among the "action sites" in the deployment environments (aspects such as lighting, specifics of the product to be examined, etc). Yet unless these sources of complexity are accepted, they block deployment specialists from duplicating success stories that the application developer may have used when creating their solution. Instead, we end up with an LCD example that will often require major changes from the AI experts' original designs. Moreover, while our examples focused on environmental and use-case diversity, we should also acknowledge that there are non-trivial resource diversity issues too.

For instance, on a shop floor, the available computational resources, like the type and power of AI accelerators, or even the number of CPU threads, can be completely different from those in the AI experts' labs, requiring that the pipeline be reengineered to run in the new environment. This will surely be more than a deployment engineer can handle, so the AI expert will have to step back in, reevaluate and change each model's hardware requirement to support the now-much-more-complex deployment needs. To make the matter more complicated, some dedicated tuning, like assigning more resources to the models working in critical data path, is needed. All these require the AI experts to be equally familiar with the deployment environments, which is a duplicate of deployment specialists' work.

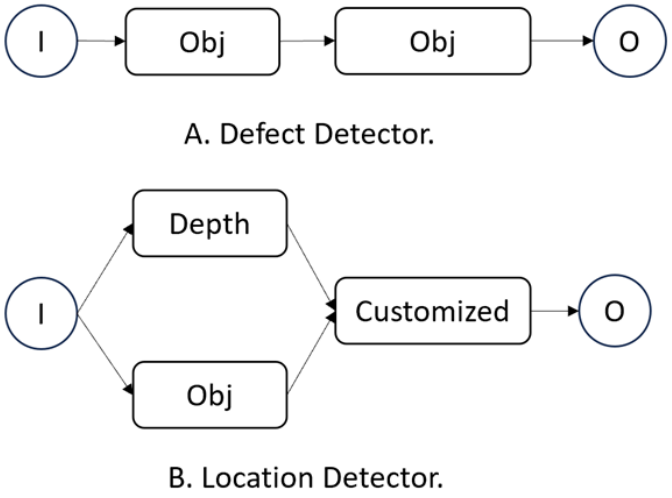

Fig. 2. Sample application data flow graph.

Dash responds to this challenge by separating the data flow logic, i.e., how the program should run, from the data placement logic, i.e., where should a piece of data or an AI model should reside, a task that is left to the Cascade scheduler (an algorithm called Navigator, which evolved from an earlier version referred to as TIDE [20]). A paper on Navigator is in preparation.

V. Experimental Use of Dash

The Dash prototype was used to design and deploy two AI applications. The first (Fig. 2-A) is a simple defect detector of the kind introduced early in the paper. The application consists of two object detection models ("obj" in the graph) concatenated in series. The first object detection model is a fine-tuned ResNet-50 model [21] to determine the parts' shapes and the second is a DETR model [22] to find the location of drilled holes on the parts. Data preprocessing and conforming logics are all added to the models.

The second (Fig. 2-B) is intended to locate a part on a conveyor belt. It consists of two models: a depth detector model ("Depth" in the data flow graph) to tell how far the item is from the camera and an object detection model ("obj") to identify the part category. The depth detection model is a fine-tuned GLPN model [23], and the object detection model is another fine-tuned DETR model. Different from defect detector, now the two models are working in parallel and their respective analyzing results are aggregated together in the user-provided aggregation logics (the "Customized" module in the graph).

Recall that Dash itself runs as a GUI on an AI expert's computer, targeting Cascade as its lowest-latency data and compute hosting framework. In the experiments we now describe Cascade is deployed on a small cluster configured to mimic a typical industrial edge server cluster; Dash interacts with Cascade as an external client, communicating over a high speed network. Our cluster has six Intel hosts, five of which are equipped with Nvidia Tesla T4 GPU to carry out the inference tasks with AI models. We installed Cascade on all six: although the sixth lacks a GPU, it can still support host computing, and the Cascade task scheduler is smart enough to not place applications dependent on GPU computing on it. All nodes have substantial memory and storage capacity, and support high-speed reliable communication (RDMA) for node to node data movement. Readers unfamiliar with this hardware can understand it as a form of TCP offloaded to hardware: it enables reliable, ordered data passing with data fetched and written directly from end-host or GPU memory via DMA. Our RDMA hardware runs at 100 Gb/s (12.5 GB/s) with one-way latency of 0.75*us*. For the experiments described here, we chose not to emulate edge clients, and instead select images from a Microsoft image database of "common objects in context" (COCO [24]), using these as application input sources (the circled "I" in Fig. 2). Industrial shop floor cameras often use 1080 pixel image width, hence we limit ourselves to color images of size 1920 pixels * 1080 pixels. Every application is tested with 200 invocations, and at the end of each run, the result is written to the output location specified by the output module (the circled "O" module in Fig. 2).

Our focus is on the reactive delay of the AI pipelines. We measured two data pathways: end to end latency, which is the total time needed to run the entire data flow logic in one invocation; and module latency, which is the time spent on each single module, including the Dash-provided AI modules and the user-provided data processing module. The results are shown in Fig. 3.

In Fig. 3, the box area represents the latency range where 50% of experiment results fall in. The lower edge of the box is the lower quartile, and the upper edge is the higher quartile. The yellow line in the box is the median value. The error bar represents the standard deviation, centering around the average value, which is the black dot in the figure. The color blue (left-most boxes) represent total end-to-end latency using the scale shown on the left Y axes. The color green (right) represents time spent in each individual module. In a real shop-floor environment we would need to add in the latency of the camera used to acquire the images and the data transfer time from camera to the edge client host, and then from edge client host to the Cascade server. We believe that 10-to-15-millisecond one-way latencies would be incurred for the input step. A further 1 to 2 milliseconds delay then could be incurred to initiate an action based on the model output, for example to instruct a robot to remove a defective part.

From discussion with domain experts, we learned that a total end-to-end latency of 400 milliseconds or less would meet typical product inspection requirements. As we can see from the results, Dash-generated applications can complete the needed computation within hundreds of milliseconds.

VI. Future Work

Our future work is directed at making Dash easier to use. To enhance user experience we are working on improving model recommendation and hyperparameter auto-fill, based on information collected from edge clients. In the future,

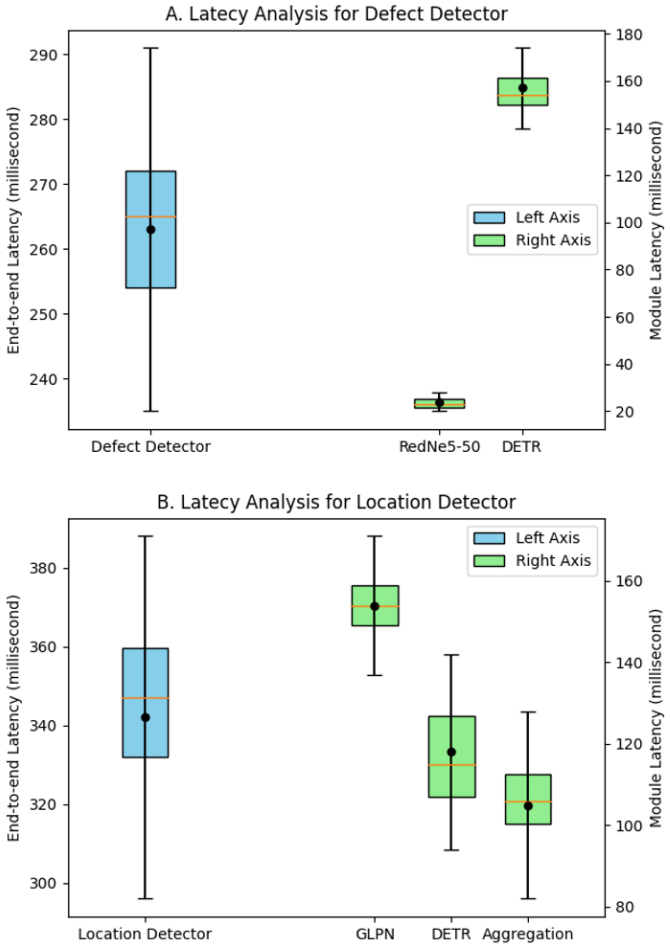

Fig. 3. Sample application latency analysis.

by leveraging powerful AI models we hope to achieve a high-quality and easy-to-ease recommendation process. The expectation is that with minimum deployment specialists' input and abundant information automatically collected from the deployment environments, Dash can efficiently populate the templates that AI experts have provided.

We are also considering other features, including enhanced "AI library" support so that an AI expert could easily select a containerized AI appropriate for a given task from a collection of options, on-the-air debugging features aimed at deployment specialists to assist them in verifying that their deployment will match requirements without needing the help of the AI expert for this task, and debugging tools to assist the team in localizing problems that do arise.

VII. Conclusions

There is tremendous demand for low-code solutions in modern computing, a reflect both of the huge scope of platforms for tasks like AI computing and cloud deployment, and a shift of CS emphasis: in many AI settings, the innovation resides in the design of the AI itself, whereas the implementation is often modular and extensively reuses existing AIs or existing computational kernels. Dash examines the idea of low code for AI (LCDAI), showing that by acknowledging the differences in roles between AI expert (who designs a kind of template for the AI application) and deployment specialist (who takes the template and then instantiates it in a concrete setting), we can improve both experiences. Doing so yields a more scalable LCDAI product, in which fewer AI experts are needed, while deployment experts are guided to more successful outcomes with less need to escalate questions back to the original AI design team. Dash is a large system and some parts are still under development, but our prototype is already demonstrating development and performance benefits.

Acknowledgment

We appreciate Tiancheng Yuan's suggestions on AI model selection. This work is supported in part by funding from Siemens Technology. Any opinions, findings, and conclusions or recommendations expressed in this material are of the author(s) and do not necessarily reflect the views of the sponsor. Additional support for our effort was received from AFRL/RY under the SWEC program and Microsoft Research.

References

[1] Nvidia. Graph composer user guide. *https://docs.nvidia.com/metropolis/deepstream/dev-guide/graphtools-docs/docs/*.

[2] Microsoft. Azure machine learning designer: visually build, test, and deploy machine learning models. *https://azure.microsoft.com/en-us/products/machine-learning/designer/*.

[3] Amazon. Amazon sagemaker canvas user guide. *https://docs.aws.amazon.com/sagemaker/*.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, and et al. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, and et al. Pytorch: an imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[6] Apache Kafka. Kafka 3.5 documentation. *https://kafka.apache.org/documentation/*.

[7] Kubernetes. Kubernetes documentation. *https://kubernetes.io/docs/home/*.

[8] Apache Yarn. Apache hadoop yarn documentation. *https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html*.

[9] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, and et al. Derecho: fast state machine replication for cloud services. *ACM Transactions on Computer Systems*, Volume 36 Issue 2 Article No. 4, 2019.

[10] L. Rosa, S. Jha, and K. Birman. DerechoDDS: efficiently leveraging rdma for fast and consistent data distribution. *6th International Workshop on Critical Automotive Applications: Robustness Safety*, 2021.

[11] W. Song, Y. Yang, T. Liu, A. Merlina, T. Garrett, and et al. Cascade: an edge computing platform for real-time machine intelligence. *ApPLIED 2022*, 2022.

[12] J. F. Arinez, Q. Chang, R.X. Gao, C. Xu, and J. Zhang. Artificial intelligence in advanced manufacturing: current status and future outlook. *Journal of Manufacturing Science and Engineering*, 142(11):110804, 08 2020.

[13] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, dec 2009.

[14] A. D. Maccormack, L. J. Newman III, and D. B. Rosenfield. The new dynamics of global manufacturing site location. *Sloan Management Review*, Vol. 35, No. 4:69–80, 1994.

[15] J. Redmon, S. Divvala, R. B. Girshick, and A. Farhadi. You only look once: unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[16] Hugging Face. State-of-the-art machine learning for pytorch, tensorflow, and jax. *https://huggingface.co/docs/transformers/index*.
[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, and et al. An image is worth 16x16 words: transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.
[18] C. Wang, A. Bochkovskiy, and H. M. Liao. Yolov7: trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, 2022.
[19] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, Y. Kwon, and et al. Ultralytics/yolov5: v7.0 - yolov5 sota realtime instance segmentation, November 2022.
[20] Y. Yang, W. Song, and K. Birman. Navigator: decentralized scheduler for latency-sensitive dag structured ml workflows. *unpublished*.
[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.
[22] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and et al. End-to-end object detection with transformers. *CoRR*, abs/2005.12872, 2020.
[23] D. Kim, W. Ga, P. Ahn, D. Joo, S. Chun, and et al. Global-local path networks for monocular depth estimation with vertical cutdepth. *CoRR*, abs/2201.07436, 2022.
[24] T. Y. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, and et al. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.