

Scaling Up Wearable Cognitive Assistance for Assembly Tasks

Roger Iyengar
raiyinga@cs.cmu.edu

October 2021

Thesis Proposal

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Mahadev Satyanarayanan (Satya) (Chair)
Martial Hebert
Roberta Klatzky
Padmanabhan Pillai (Intel Labs)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2021 Roger Iyengar
raiyinga@cs.cmu.edu

Contents

1	Introduction	1
1.1	Thesis Statement	2
1.2	Related Work	2
2	State Detection	4
2.1	Related Work	5
2.2	Preliminary Work	6
2.2.1	Stirling Engine	6
2.2.2	Realtime Data	7
2.2.3	Improving Accuracy	7
2.2.4	Hierarchical Decomposition	8
2.3	Proposed Work	9
3	Escalation to Human Experts	10
3.1	Related Work	11
3.2	Preliminary Work	11
3.3	Proposed Work	12
4	Synthetic Data	13
4.1	Related Work	14
4.2	Preliminary Work	14
4.3	Proposed Work	15
5	Runtime Platform	16
5.1	Motivation	16
5.2	Implementation Details	17
5.2.1	Flow Control	17
5.2.2	Components	18
6	Expected Timeline	19
6.1	Completed Work	19
6.2	Remaining Work	19

1 Introduction

Wearable Cognitive Assistance (WCA) applications have been developed to help users with tasks such as assembling physical objects, remembering people’s names, exercising, and playing games. Table 1 lists some examples of WCA applications that have been developed. These applications utilize mobile devices, such as smart glasses or a smartphone, to capture data and interact with the user. Data is captured using sensors such as cameras. These cameras may be mounted on glasses that the user wears, or held in a tripod with a view of the user’s workspace. Feedback is provided in the form of synthesized speech and images shown on the display of the mobile device.

WCA is a compelling use case for edge computing. Many of these applications utilize large deep neural network (DNN) models that are too computationally intensive to run on a small and lightweight mobile device. However, these applications generate large volumes of data that must be processed quickly. Therefore, computation must be offloaded to a server with close network proximity to the mobile device that is capturing the data [43]. We will henceforth refer to this server as a cloudlet.

The work I propose builds on top of earlier research on WCA. Ha et al. [27] introduced the first version of a programming framework called Gabriel. This framework includes networking and runtime components for WCA applications. Chen et al. [19] developed an initial set of WCA applications, determined how much latency was acceptable for these applications, and examined how changes to the network, hardware, and algorithms used can affect end to end latency. Wang et al. [50] evaluated strategies to allow multiple users to share a single cloudlet. The requirement that a mobile device must have close network proximity to a cloudlet makes cloudlets a shared resource. Sharing cloudlets among multiple users is therefore a requirement for a large number of people to be able to use WCA applications. Pham et al. [39] developed a toolchain that allows people to develop WCA applications without writing any code.

This proposal focuses on WCA applications that help users complete physical assembly tasks. Users are given step by step instructions, which requires the application to determine when a user has completed a step of the task. Applications have been developed for a lego kit [9], a lamp [8], and a toy sandwich [10]. These tasks all required fewer than ten steps and used parts that had distinct shapes and colors. Taking WCA applications to the next level will require supporting tasks with many more parts, many more steps, parts that are small relative to the full object being assembled, and a combinatorial explosion of error states. We propose to address these challenges in this research.

One significant challenge is the amount of labeled data that is required for training computer vision models. Each step of the task, and every error state that the developer would like to detect, must be represented in the data that the models get trained on. Increasing the number of steps thus directly increases the amount of data that is required for training the models. Collecting and labeling all of this data is an incredibly time-consuming process. This process can take up to 2 or 3 hours per task step. Developing these models is an iterative process, which involves testing under different lighting conditions, and then collecting more data in environments that the model performs poorly in. Creating training sets is thus a significant barrier to developing WCA applications that support large numbers of steps and large numbers of parts.

As the number of parts being assembled increases, the size of the final object does as well. An

object made out of 100 parts is going to be significantly larger than some of the individual parts. The camera is going to capture images of the full object, but the application needs to determine when each individual part has been inserted into the correct location on the full assembly.

Machine Learning models for computer vision require training data for each object that the model will detect. Detecting every possible error state using such a model would require collecting and labeling data for each of these possible states.

These challenges lead to the following thesis statement.

1.1 Thesis Statement

Scaling up WCA to tasks with 100 parts or more is challenging because of (a) the difficulty of vision-based state detection with very small parts in the context of much larger objects being assembled; (b) the combinatorial explosion of possible error states; and (c) the large manual effort needed to create accurate DNNs that can reliably determine when task steps have been completed. These problems can be solved by a combination of (1) hierarchical decomposition of complex assemblies into modular compositions of subassemblies, (2) on-demand seamless escalation for live expert assistance, and (3) synthetic generation of training sets for born-digital components. The resulting solution can be implemented in a scalable and maintainable way using modular software components. This will enable the development of WCA applications for more complex tasks, which is a necessary step along the path towards making WCA applications practical for real world tasks.

1.2 Related Work

We have found a large body of work on systems to aid with assembly tasks. However, none of these systems determined when steps were completed using computer vision models that processed data from an RGB camera. The techniques we propose do not require instrumenting parts of workspaces with sensors. Our system also does not require the person who is completing the task to determine when a step has been completed, and then indicate this to the system. On top of this, our system only starts a call with a task expert when the person completing the task has reached an error that the computer vision models cannot handle. This burdens task experts much less than a system where the task expert must be on a call to help a user with the full task.

Fraser et al. [23] developed a system to distribute steps for an assembly task among members of a group completing the task together. Instructions are displayed on a smartphone screen, and users manually press a button to indicate when a step is completed. The authors ran a user study where people assembled an IKEA cabinet and a Meccano bridge kit. Requiring people completing the task to indicate when a step is completed creates an additional burden that our system avoids. Antifakos et al. [16] developed a system that determines when steps for assembling an Ikea wardrobe have been completed, using sensors such as gyroscopes, accelerometers, force sensing resistors, and infrared distance meters. This system allows users to complete steps of the task in different orders. However, their system was purely focused on detecting completed steps, and it did not give the users any instructions. Gupta et al. [25] used a Kinect sensor to guide users through assembling objects out of Duplo blocks. The system determined when steps were completed by processing data from the Kinect sensor. Duplo blocks have bright colors and

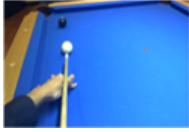

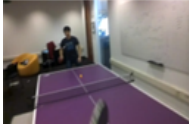







App Name	Example Input Video Frame	Description	Symbolic Representation	Example Guidance
Pool		Helps a novice pool player aim correctly. Gives continuous visual feedback (left arrow, right arrow, or thumbs up) as the user turns his cue stick. The symbolic representation describes the positions of the balls, target pocket, and the top and bottom of cue stick.	<Pocket, object ball, cue ball, cue top, cue bottom>	
Ping-pong		Tells novice to hit ball to the left or right, depending on which is more likely to beat opponent. Uses color, line and optical-flow based motion detection to detect ball, table, and opponent. Video URL: https://youtu.be/_lp32sowyUA	<InRally, ball position, opponent position>	“Left!”
Work-out		Counts out repetitions in physical exercises. Classification is done using Volumetric Template Matching on a 10-15 frame video segment. A poorly-performed repetition is classified as a distinct type of exercise (e.g. “good pushup” versus “bad pushup”).	<Action, count>	“8”
Face		Jogs your memory on a familiar face whose name you cannot recall. Detects and extracts a tightly-cropped image of each face, and then applies a state-of-art face recognizer. Whispers the name of the person recognized.	ASCII text of name	“Barack Obama”
Lego		Guides a user in assembling 2D Lego models. The symbolic representation is a matrix representing color for each brick. Video URL: https://youtu.be/7L9U-n29abg	[[0, 2, 1, 1], [0, 2, 1, 6], [2, 2, 2, 2]]	“Put a 1x3 green piece on top”
Draw		Helps a user to sketch better. Builds on third-party app for desktops. Our implementation preserves the back-end logic. A Glass-based front-end allows a user to use any drawing surface and instrument. Displays the error alignment in sketch on Glass. Video URL: https://youtu.be/nuQpPtVJC6o		
Sandwich		Helps a cooking novice prepare sandwiches according to a recipe. Since real food is perishable, we use a food toy with plastic ingredients. Object detection uses Faster-RCNN deep neural net approach. [41] Video URL: https://youtu.be/USakPP45WvM	Object: “E.g. Lettuce on top of ham and bread”	“Put a piece of bread on the lettuce”

Table 1: Example Wearable Cognitive Assistance Applications. The input frame is sent to a cloudlet, which converts it to the symbolic representation. The symbolic representation is then used to give guidance. (Source: Adapted from Satyanarayanan [42])

Prior Work	Detection of Step Completion	Other Attributes
Fraser et al. [23]	User presses a button	Multiple users working on task together.
Antifakos et al. [16]	Automatic, using gyroscopes, accelerometers, force sensing resistors, and more	Requires sensors to be installed in the Ikea parts. Did not give instructions.
Gupta et al. [25]	Automatic, using Microsoft Kinect	Builds a full virtual model of what the user has constructed. Only supports large colorful duplo blocks.
Aehnelt and Urban [15]	Automatic, using RFIDs and infrared light barriers	Requires sensors to be present in working environment.
Lafreniere et al. [34]	Manually from user	The final assembled object was massive. Many parts were identical.
Johnson et al. [31]	Remote task expert	Compared Tablet with Google Glass.

Table 2: Existing systems that helped users with physical assembly tasks.

simpler shapes than the objects that our tasks use. Aehnelt and Urban [15] developed a system that provides instructions to users on smartwatch displays, and determines when steps have been completed using sensors that are typically present in a factory environment, such as RFIDs and infrared light barriers [17]. Our system does not require the presence of such sensors.

Lafreniere et al. [34]’s system guided users through assembling a pavilion out of bamboo sticks. The users moved around a room, picking up certain sticks and then delivering them to a robotic arm. Users wore Apple Watches, and were given written instructions on the watches’ displays. A user swiped on the face to see the next instruction. The locations of each user were also tracked using BLE beacons that communicated with iPhones that users carried. A centralized system used the location information to decide the next instruction that should be given to each user. This system requires users to manually request the next instruction, while our system automatically gives the next instruction after the user completes a step.

Johnson et al. [31] examined systems where remote experts helped users with assembly tasks. The people completing tasks communicated using a tablet for one task, and a Google Glass for another. In addition, they looked at a case where the user could stay in one place and a case where the user had to move across different workspaces to complete the task.

Table 2 provides a brief summary of all of the existing systems described in this section.

2 State Detection

WCA applications for physical assembly tasks must detect when steps of the task have been completed. A user can either wear a device like Google Glass or mount a smartphone on a tripod and position the camera to see the object that they are assembling. For ease of exposition, our discussion will focus on a tripod-mounted smartphone. The smartphone gives the user an instruction, waits for them to complete this, and then gives them the next instruction. Completing a step might require adding a part to the assembly, removing a part from the assembly, or repositioning

the object or the camera to match a certain view. The application must determine when a step is completed based on images from the camera. Avoiding false positives is important because they will cause the application to give the user a new instruction before the previous step gets completed.

2.1 Related Work

Existing computer vision models and techniques can be leveraged for use in wearable cognitive assistance applications. The work in this proposal will focus on developing systems that use existing computer vision algorithms, and we do not anticipate the need to develop any computer vision algorithms ourselves.

Gebru et al. [24] used a two step process to find and distinguish cars that appeared in Google Street view images. Their first step was finding the regions of images that were likely to contain cars. Then their second step was classifying the type of car in that region. We will train models on our own new data, but these models will use existing neural architectures.

Image classifiers give a single class label for a full image, and do not provide a bounding box. These are most useful when a scene has already been cropped to a region involving a single object. Imagenet [20] is a classification dataset which contains classes for 1000 objects. Object detectors provide bounding boxes and labels for objects in an image. This is particularly helpful when an object might only take up part of the camera view, or there might be multiple objects visible at once. Microsoft COCO [36] is an object detection dataset with 80 classes.

Faster R-CNN [41] is an object detector that is used in our lamp [8] and toy sandwich [10] assembly assistants.

Imagenet has the classes “race car,” “sports car,” and “streetcar,” in addition to classes for objects that aren’t cars. The Stanford Cars dataset [33] is more fine-grained, with images of cars that are labeled with the year, make, and model. The Fast MPN-COV [35] image classifier performs well on several fine-grained classification datasets.

Object detectors have been used for finding screws in pictures of aircraft parts [37] and consumer electronics [22]. Wu et al. [51] found differences between book covers using a modified version of Faster R-CNN.

Simon [44] argued that all complex systems are made up of smaller systems. These smaller systems are made up of even smaller systems, thus forming a hierarchy with several layers. Reasoning about a smaller system on its own is easier than trying to understand a full system all at once. We can apply this idea to state detection for WCA applications by decomposing a large assembly task into separate sub-assemblies with 8-10 parts. This limits the scope of what any one computer vision model that we use is responsible for. It also allows multiple developers to work on computer vision models for different parts of the task completely independently from each other. Finally, it also simplifies performance of a task over an extended period (e.g. multiple days). A person who has to stop work in the middle of a task will have an easier time if the task is split into subtasks that don’t require any context from earlier subtasks. They can start work from the beginning of a subtask every time, rather than having to recall anything about steps they completed the last they they worked on the task.

2.2 Preliminary Work

Our early applications used large objects that had distinct colors or shapes. These objects were easy to see in an image of the full object that was being assembled. However, as we scale up to large assemblies that are less toys and more realistic we need to detect small parts like screws that take up a small section of the object being assembled. Asking the user to move the camera and zoom in on parts makes WCA applications cumbersome to use. When something is occluded, it becomes necessary to reposition objects or the camera. However, we want to minimize the number of times the user has to do this. We therefore need to be able to detect the presence or absence of a single part in a large assembly.

2.2.1 Stirling Engine

We developed a WCA application that guides users through disassembling a Stirling engine. We have posted a video [12] of this application and released the code [11]. This task requires 22 steps. All of the parts are made out of metal, with the exception of one ring that is made out of silicone. Some steps just require removing a single screw, and the engine looks very similar before and after this step has been completed. Figure 1 contains some examples.

The task was broken apart into sub-assemblies, and we trained an object detector and an image classifier for each sub-assembly separately. Our application uses Faster R-CNN to find the region of an image that contains the relevant sub-assembly for a certain step of the task. It then crops the original image around this region. This cropped image is then classified using Fast MPN-COV [35]. Our Fast MPN-COV model achieved 91% top-1 accuracy on our test data. Gebru et al. [24] also found regions of interest, and then ran a separate model to classify these regions.

Figure 1 shows four of the steps required for the Stirling engine task. We labeled the images for these steps based on the number of black screws that were visible, rather than using a different label for each step. For example the first and third steps in the figure were both given the label “2 Black Screws.” The training script for Fast MPN-COV randomly flips images horizontally, so we wanted to train the model to differentiate images based on the number of screws rather than the orientation of objects. The initial steps for this task all require removing screws or flipping the engine to show screws that were previously occluded. Therefore, every step changes the number of screws that are visible.

Grouping separate steps into single labels is an effective way to simplify the computer vision problem. This technique is generalizable to any task. However, steps that share labels are indistinguishable to the image classifier. This is not a problem when consecutive steps all have different labels. If completing a set of steps requires going from label A, to label B, and then back to label A, the application can see that these two transitions happened. However, the application would have no way of detecting if a transition from label A to label A occurred. The classifier needs to see some change in label. Any authoring tool for WCA applications must require that consecutive task steps do not share the same computer vision class label. However, steps that are not consecutive can share the same computer vision class label.

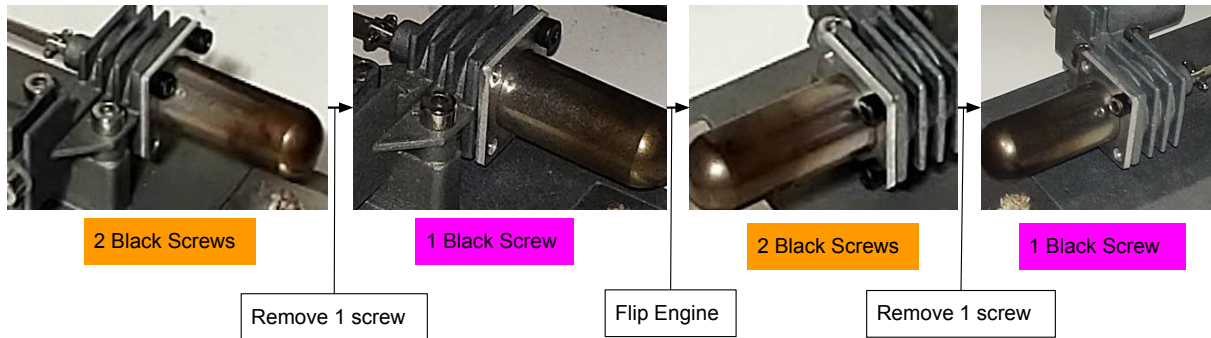


Figure 1: Four states from our WCA application for a Stirling engine. The steps look visually similar aside from the number of screws that are visible. The text in colored boxes are the labels that our image classifier was trained on. Note that some different steps were given the same label, but consecutive steps must have different labels. The text in the white boxes describes the actions users take to transition between these states.

2.2.2 Realtime Data

The code for many computer vision models are written to run inference on batches of images that are stored on disk. The torchvision package contains functions for loading images from disk, in batches. Using these models in WCA applications requires modifying the code to run the models on images being transmitted over the network, one by one. The input batch size must be set to 1, because anything larger would require building up a queue of images that would be run through the model together as a single input. A larger batch size would improve the frame rate for inference, but hurt the latency.

Live data must be as similar as possible to the data that the models are trained on. For example, converting a JPEG image to raw pixel values using OpenCV will result in slightly different values than using Pillow will. We observed that our Fast MPN-COV model performed significantly better with images opened using Pillow than with images opened using OpenCV. The training images were opened using Pillow, but we did not expect opening JPEGs with OpenCV and Pillow to result in different color values.

Processing images while a user is in the middle of a step wastes bandwidth and computing resources on cloudlets, and it might lead to an application mistakenly believing that a step has been completed before it actually has been. Our application discards frames that have identical perceptual hashes as Hu et al. [30] suggest. This reliably removes frames while the user’s hands are moving.

2.2.3 Improving Accuracy

We ran into several issues that hurt the accuracy of our models. First, the resolution of the images that the smartphone was capturing was too low. We increased the resolution to 1920x1080 pixels, which is the highest resolution that Android CameraX’s ImageAnalysis use case supports. These images are then cropped around the region of interest that Faster R-CNN finds. The cropped image is resized to 448x448 pixels before it is classified by Fast MPN-COV. We observed that

the cropped images were at least 448x448 pixels after we set the original resolution to 1920x1080 pixels.

We also saw an improvement in performance after removing images that had the same perceptual hash value from our train, validation, and test sets. We made sure that all images (across all three sets) had unique perceptual hash values.

Lastly, we placed a lamp on the table, in between the user and the object being assembled, as shown in Figure 2. We found that the Essential Phone’s built in flash was too dim to illuminate objects well, but the lamp made a difference.

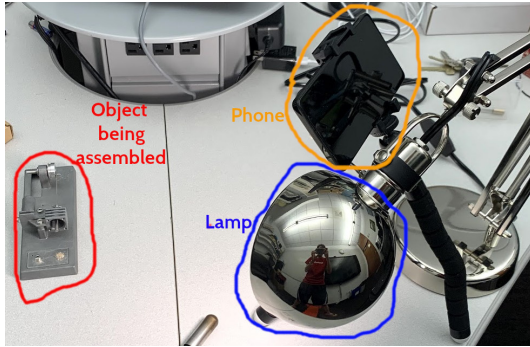


Figure 2: The placement of our lamp

2.2.4 Hierarchical Decomposition

An object made up of 100 parts is going to be larger than most of the individual parts. Rather than asking the user to move the camera and zoom in on each part they install, the system should be able to determine when a step has been completed based on an image that contains most or all of the full object being assembled. Breaking up a large object into a collection of sub-assemblies makes this possible. Our system uses a two stage process where it first finds the region of an image that contains the sub-assembly involved in a step. It then crops the image around this region, and the next model determines if the step has been completed based on the cropped image. Inspired by Simon’s argument that all complex systems are made up of smaller systems [44], we argue that any object that is assembled from more than ten parts can be decomposed into sub-assemblies. Hence the hierarchical approach proposed here can be scaled upwards, without obvious limits.

The Stirling engine task from Section 2.2.1 requires 22 steps. Figure 3 shows two of the sub-assemblies. The application uses a different computer vision pipeline for each sub-assembly. The code selects the correct pipeline based on the current step that the user is working on. Each step involves removing a piece from one sub-assembly. None of the sub-assemblies involve more than 8 steps. Splitting the task up into sub-assemblies thus simplifies the scope of the problem to developing a set of assistants for 8 step tasks.

The number of steps required for each sub-assembly is not something that we have any fixed rules about. We have found that limiting the number of steps to 10 appears to work well empirically, but the optimal number of steps for a sub-assembly will likely vary based on the task.

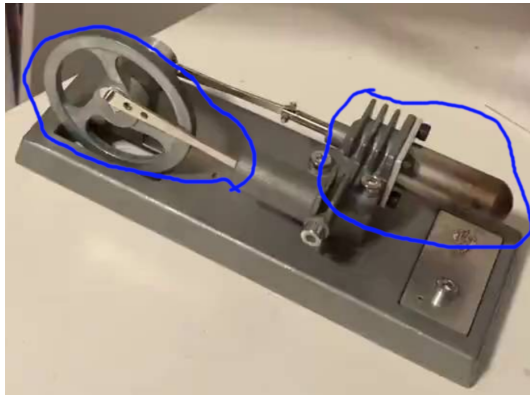
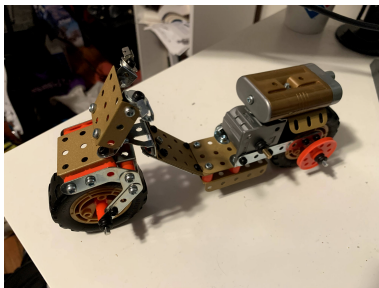


Figure 3: A stirling engine with two sub-assemblies highlighted

Figure 4 shows how a model motorcycle can be broken up into three sub-assemblies. The Stirling engine has a single large base, but the motorcycle is simply assembled from small pieces. Therefore, assembling the motorcycle would require an additional set of steps at the end, to connect the sub-assemblies together.



(a) The fully-assembled model



(b) Sub-assemblies

Figure 4: A model motorcycle from a Meccano Erector kit

2.3 Proposed Work

These techniques worked well for our Stirling engine application. We now need to validate that they will also support other tasks. I plan to develop two additional applications for other tasks

that involve small parts and require more than 20 steps. If the techniques used for the Stirling engine application also work for these additional tasks, this will provide additional validation of the techniques.

I will expand the Ajalon tools to support these new techniques. OpenTPOD uses an old version of TensorFlow, and only works with a single object detector. I will update OpenTPOD to use the newest version of TensorFlow, and support techniques required for more realistic tasks. Ajalon only supports the use of a single object detector. This is insufficient for tasks like the Stirling engine that have parts that take up a small region of the full assembly. We will extend Ajalon’s functionality to support training fine-grained image classifiers in addition to object detectors. This will enable applications built using Ajalon to leverage the two-stage process we used for our Stirling engine assistant. Note that the two-stage process refers to first finding a the region of an image containing a sub-assembly, and then classifying just that region. It is applicable to tasks with multiple levels of sub-assembly hierarchies.

An assistant for a task involving multiple sub-assemblies is effectively a series of independent applications. Once the user completes one sub-assembly, they should automatically be taken to the assistant for the next one. If the sub-assemblies must be connected together after that, there will be an assistant for these steps as well. We will add functionality to Ajalon so that developers can specify a series of sub-assemblies required for a task. When a person completing a task uses the assistant, it will automatically start giving them instructions for the next sub-assembly after they have completed the previous one. Tasks can be broken into sub-assemblies the same way that long documents can be broken into chapters, sections, and subsections. Sub-assemblies near the top of the hierarchy are going to be made up of multiple levels of sub-assemblies below them.

3 Escalation to Human Experts

The techniques presented in the previous sections allow WCA applications to detect states that the developer trains models to handle. The developer can provide example images of the object after each step has been done correctly. However, there are many possible ways that an object can be put together incorrectly. It is not possible to collect images of every mistake that someone completing a task might make. People using these applications in the real world are going to reach some states that the models were not trained for. As Dr. Reynold Xin once said, “A machine learning model is only as good as the data it is fed [38].” Our models can signal to our application that an image “looks most similar to this set of images from the training data.” These models are not capable of a more general understanding, such as “the long brass piece is screwed on upside down.”

Detecting all possible error states would require us to have examples of these states in our training data. There is a combinatorial explosion in the number of error states, compared to the number of correct states, so collecting training data for every possible error is not practical. Instead, our applications handle errors by connecting the person completing the task to a human who is an expert on this task.

3.1 Related Work

Microsoft Dynamics 365 Remote Assist [3] lets a human task expert walk someone through every step of a task. This is done remotely with the help of a headset, but the expert must help the person one on one, through the entire task. A task expert's time is valuable. We believe that integrating wearable cognitive assistance with human assistance will allow products like Microsoft's to scale from requiring experts to work one on one through each step of the task to enabling experts to help multiple users concurrently.

3.2 Preliminary Work

The computer vision models that our applications use are not perfect. In order to handle cases where a model makes a mistake, our applications allow the user to start a call with a human who is an expert on the task being completed. The human expert sees a feed from the user's camera and they can talk the user through correcting problems. In addition, the expert can change the step that the application thinks the user is in the middle of, and then the user can continue to receive guidance from the application after the call ends.

Computer vision models can recognize objects that they have been trained on. When we train models for Gabriel applications, we consider each state of a task to be one object. Open World Object Detection [32] is an active area of research into models that can learn to detect previously unlabeled objects. However, this does not help with recognizing such an object the first time it is seen. Gabriel applications need to handle all errors, even ones that have not been seen before. A human who is an expert on the task can do this.

Correcting error states in Gabriel applications can be done on the order of tens of seconds to a few minutes, unlike driving a car which might require sub-second response times. It's perfectly acceptable for the user to press a button to call for help from an expert, if the application does not detect that a step has been completed after a certain amount of time. The user will then be connected to someone who is an expert on this task. The expert will see the camera feed from the headset and talk back and forth with the user to help them get back to a state that the computer vision models can handle. The expert will also have the ability to update the application's state, so the user can continue receiving automated guidance from an earlier or later step after the call.

We connect users to task experts using Zoom, which offers SDKs for several platforms [1]. The Gabriel user runs an Android application on a smartphone, which starts a call with the expert using Zoom's Android SDK. We plan to develop this application to work on a Google Glass as well. The human expert uses a web application that incorporates Zoom's Web SDK. Figure 5 shows a screenshot of this application. The code can be modified to use a different video calling service in the future. The components of the system are shown in Figure 6.

We would like to develop a version of the client application for Microsoft Hololens. However, this might require us to change to a different video calling service if Zoom's SDK does not run on the Hololens.

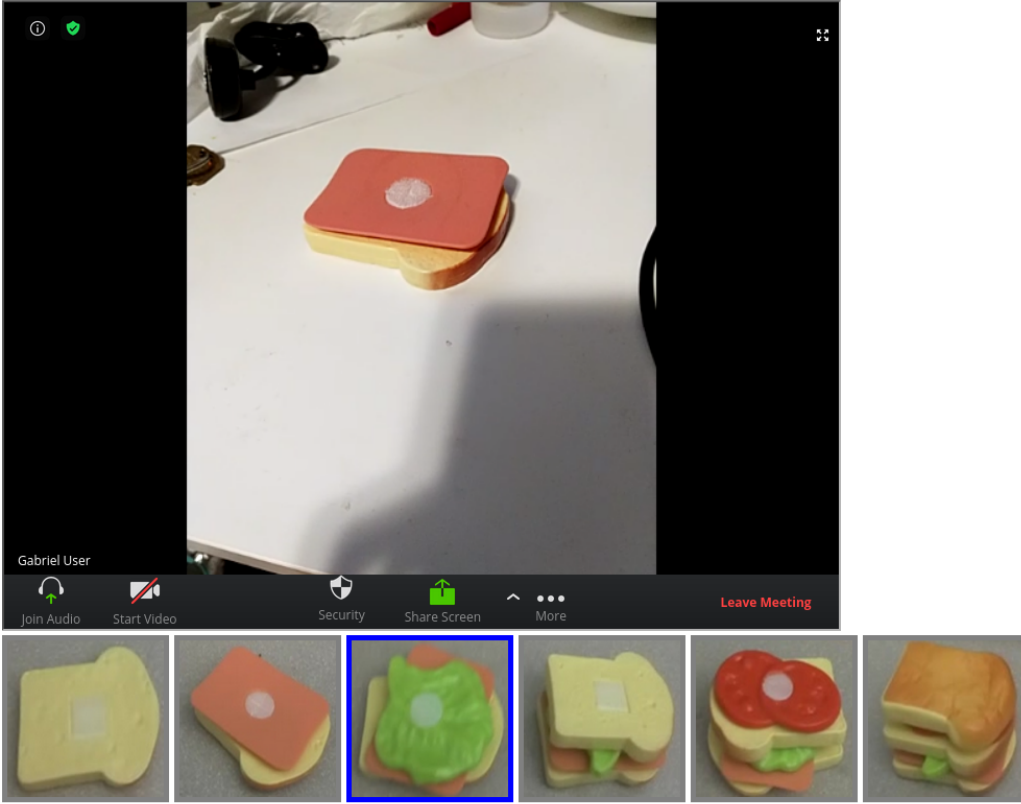


Figure 5: A screenshot of the web application used by the human task expert. The feed from the user’s camera is shown on top. The task steps are shown on the bottom. The application’s current state is the step surrounded by the blue box. Clicking on a different step will change the application’s state to the one that was clicked on.

3.3 Proposed Work

Error states might be caused by the person completing the task making a mistake, a part being manufactured differently, or a part breaking. Regardless of the reason, we would like to improve our applications to handle common error states on their own, in the future. The first step to accomplishing this is having the person completing the task press a button to escalate to a task expert, as a “page fault.” The call with the task expert will be recorded with Zoom. The task expert will guide the person who was completing the task through moving their camera around, to capture multiple angles of the new error state. The application developer can access this recording at a later date, and then use it to train a new model that can recognize this error state in the future. In this way, the WCA application becomes more clever over time, and this lessens the burden on the expert.

Task experts are a limited resource. A large deployment of a WCA application would likely be used by more people completing the task than there would be available task experts. Therefore, if many people request help from task experts at once, some will have to wait in a queue until one of the experts is available. This leads to several choices that someone would have to

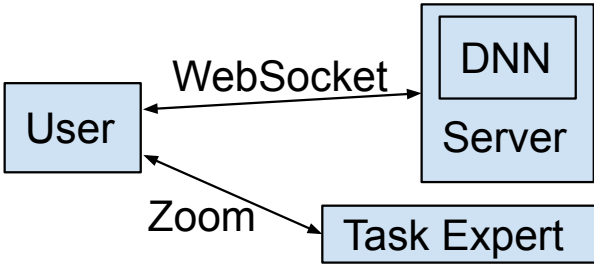


Figure 6: The components of a Gabriel application with human assistance. The user primarily receives guidance from a server running a DNN. If the user reaches a point where the automated assistance fails, they can switch over to receiving guidance from a human expert over a video call.

make in order to deploy this system at scale. How should people completing the task be removed from the queue? Calls could be answered in the order that they are received, people who have been stuck on a step longer could be given higher priority, or people could be given priority based on the number of steps they have completed. How should the number of experts scale with the number of users? It’s wasteful to have experts sitting idle, but it’s also a waste to have users stuck waiting a long time for experts.

We will explore answers to those questions through Monte Carlo simulations. These will model users attempting to complete a task using our system, getting stuck on a step, attempting to fix the problem themselves, and then requesting help from the expert. If all of the experts are busy helping other people, this person will have to wait until there is an expert available to help them.

Our first simulation will determine the total amount of time spent completing the task, across all people that did it. We will see how this value changes as we vary the number of people doing the task, the time required to complete each step, the probability of a person getting stuck on a step, the likelihood that this person will be able to figure out this problem without calling the expert, the user’s patience, and the strategy that we use to queue calls.

Our second simulation will determine the number of experts that should be available to take calls. We will observe how this changes, as we vary the maximum acceptable wait time, in addition to all of the parameters and distributions that we varied in our first simulation.

These simulations will be tools that future researchers can run using different parameter and distribution settings.

4 Synthetic Data

Collecting and labeling data to train object detectors and image classifiers is a labor intensive process. Eliminating this step is a requirement for making it practical to develop WCA applications.

4.1 Related Work

Hinterstoisser et al. [29] trained an object detector on synthetic data that outperformed an object detector trained on real data. They generated backgrounds cluttered with distractor objects. In addition, they added some distractor objects to the foreground and varied the lighting conditions that were used to render each of the images. These images looked 3D, but they were not photo-realistic. Other works have generated photo-realistic images to use as training data [26, 47], or used real background images [28, 40, 46]. Dwibedi et al. [21] avoided rendering 3D graphics altogether by cropping objects from photographs, and pasting these crops into other photographs.

4.2 Preliminary Work

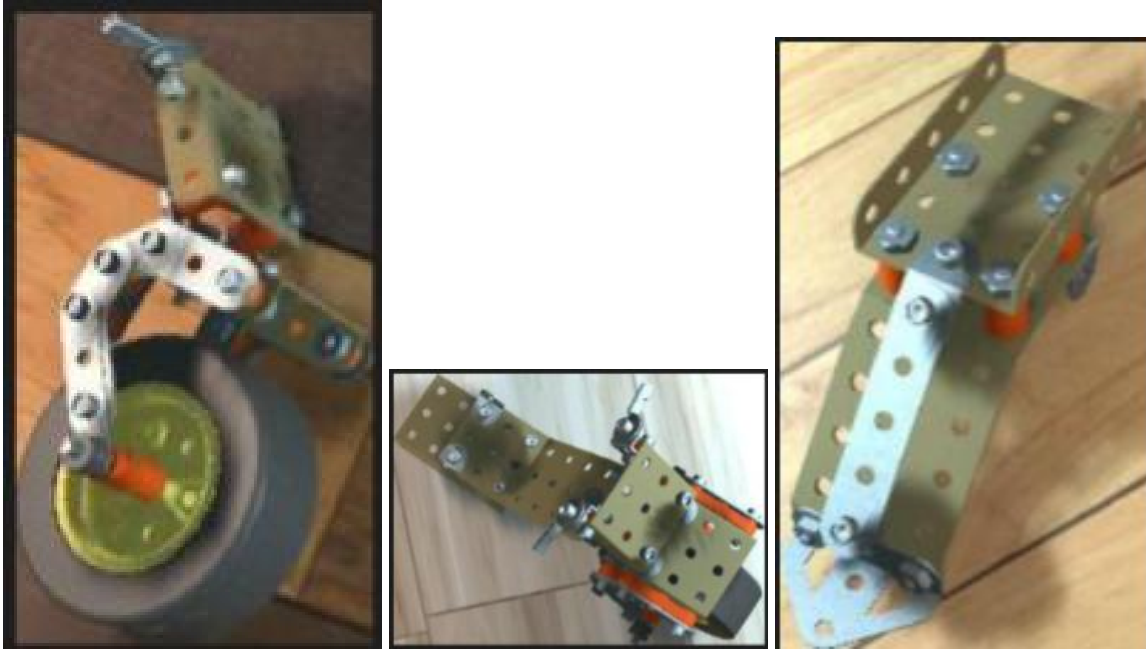
We trained object detectors to recognize sub-assemblies of a model motorcycle from a Meccano Erector kit, using synthetic and real data. The synthetic data was rendered using Unity, with backgrounds from Adobe Stock Photos. We attempted to make the images look photo-realistic. Figure 7 shows some examples of our training data.

One model was trained entirely on real data, while the other was trained entirely on synthetic data. We tested both models on a separate set of real data. The results are listed in Table 3. Object detection results are evaluated at different intersection over union (IoU) thresholds. IoU is a measure of overlap between the ground truth bounding box and then box predicted by an object detector. Increasing the IoU threshold thus requires the predicted box to be closer to the ground truth box, in order to be considered correct. The average precision of both models was perfect with an IoU threshold of 0.5, but the model trained on synthetic data only achieved 0.22 average precision with an IoU threshold of 0.75. We hope to close this gap by making our synthetic data look more realistic.

We also trained classifiers to label cropped images of parts from the Meccano kit. The classifiers trained on both real and synthetic images achieved around 0.85 Top-1 Accuracy.

	AP 0.5 IoU	AP 0.75 IoU
Real	1	1
Synthetic	1	0.22

Table 3: Object Detection test set results from Faster R-CNN models trained on real and synthetic images of parts from a Meccano kit. We are using Faster R-CNN to localize objects in images, but we do not use it to classify these objects. Thus, in our training data, we use the same class label for every bounding box, regardless of which object the box is drawn around. AP stands for average precision. This is the fraction of bounding boxes that were true positives, averaged across all confidence score thresholds. IoU is the fraction of the predicted and ground truth bounding boxes that overlap. These results are encouraging, but the AP 0.75 IoU of the model trained on synthetic data tells us that this model does not perform well enough to be used in a WCA application.



(a) Synthetic Images



(b) Real Images

Figure 7: Real and Synthetic data for a model motorcycle

4.3 Proposed Work

Our synthetic data for the Erector kit was generated using CAD files from the website GrabCad. These files were likely created by someone trying to reproduce parts from the kit, rather than anything used to make the physical parts. Thus there are some differences between the CAD models and the real physical objects. We plan to train object detectors for objects that were manufactured from CAD files that we have access to. We will henceforth refer to such objects as being “born digitally.” The objects will be manufactured to match the CAD file using a Computer numerical control (CNC) mill or a 3D printer. The physical object is a physical twin to the CAD file, which is the digital twin. We will generate synthetic data using the digital twin. This should

	Top-1 Accuracy
Real	0.86
Synthetic	0.85

Table 4: Classification test set results from Fast MPN-COV models that were trained on crops of the images that our Faster R-CNN models were trained on. For comparison, Fast MPN-COV achieved a top-1 accuracy of 0.88 on a dataset of birds [49], 0.9 on a dataset of aircrafts [48], and 0.93 on a dataset of cars [33].

reduce the variations between how objects appear in our synthetically-generated training data and how the physical twins appear in real life. We will refer to the computer vision models that we train to recognize the object, using synthetic data, as a “digital triplet.”

We will attempt to improve the robustness of our models by varying the backgrounds, lighting and camera positions when rendering synthetic images. We aim to have a variety of realistic backgrounds in these images.

We will explore the technique proposed by Dwibedi et al. [21]. They observed good performance using objects from the BigBIRD dataset [45]. We will see how their approach performs with small objects like screws, or parts from the Meccano kit. Their approach does not require CAD models or rendering 3D graphics.

5 Runtime Platform

We developed a set of software libraries for WCA applications, called Gabriel [5]. The primary feature of these libraries is to transmit data from mobile devices to cloudlets. WCA applications require responses shortly after a user completes a step, so we always want to process the newest frame possible. We never want to build up a queue of stale data to process. The library accomplishes this using a flow control mechanism similar to the one proposed by Ha et al. [27].

5.1 Motivation

The library we developed replaces an earlier implementation. The code for this earlier implementation had become unmanageable. It was tightly coupled around sending single image frames, and we wanted the ability to send chunks of consecutive frames. We also needed multiple clients to share one cloudlet, which the old code did not support. Developing a new version of the platform allowed us to use modern technologies such as Python 3, WebSockets, and asyncio. A key goal with the new version of the platform was making it easy to work with. We published server and client libraries to package repositories, so that developers can easily include them in Python and Android code. Our code includes a special case for Gabriel workflows that involve a single cognitive engine, which allows this engine to be run with the server in the same Python program.

5.2 Implementation Details

We use the abstractions of “sources” and “cognitive engines.” A source is anything that produces data on a mobile device. It could be a stream from a sensor such as a camera or microphone. A source might also be a filter that runs on the device, analyzes all frames produced by a sensor, but then only forwards some of these frames to the cloudlet. We use the term “early discard” to refer to filters like this. A cognitive engine runs on a cloudlet and processes data. A cognitive engine will process one frame of data at a time. A frame could be a single image, a short clip of audio and/or video, or set of readings from a different type of sensor.

All of the wearable cognitive assistance applications we have developed just have a single cognitive engine processing images from a single camera source. However, our framework supports workloads with multiple sources and multiple cognitive engines. Multiple cognitive engines may consume data from the same source, but we restrict each cognitive engine to consuming data from one source. This reduces the complexity of cognitive engines.

Cognitive engines are all implemented in Python. Developers implement a single function that takes a frame as its input parameter and returns a list of results when it completes. Cognitive engines that do not need to return results to mobile devices can just return an empty list.

5.2.1 Flow Control

Our flow control mechanism is based on tokens. Clients have a set of tokens for every source. When a client sends a frame to the cloudlet, it gives up a token for this source. The cloudlet returns the relevant token when the function processing the frame returns. A client will drop all frames from a source, until it gets a token for this source. Clients and cloudlets communicate using the The WebSocket Protocol [13], which is built on TCP. Therefore, tokens will never be lost due to packet loss.

Applications that are very latency sensitive, such as wearable cognitive assistance, will be run with a single token per source. Applications that can tolerate higher latency can be run with more tokens. Multiple tokens will allow frames to be transmitted while the cloudlet is busy processing other frames. This may cause frames to be buffered on the cloudlet, if the cognitive engine takes a long time to process earlier frames. As a result, there might be a significant amount of time between when a frame is captured and when it gets processed. However, using multiple tokens does avoid periods where the cloudlet does not have any frames to process because it is waiting for the next frame to be sent over the network. Increasing the number of tokens thus increases the possible delay before a frame gets processed but reduces the amount of time the cloudlet has no frames to process, when network latency is high. The number of tokens is thus a parameter that will increase the framerate for applications that can tolerate higher latency.

When multiple cognitive engines consume frames from the same source, the token for a frame is returned when the first cognitive engine finishes processing the frame. A Client will only receive a result from the first cognitive engine that finishes processing a frame, and it will not get additional results or tokens when other engines finish processing the same frame. Our server library keeps a queue of input frames for each source. When multiple clients produce frames from the same source, such as two smartphones both capturing images with a camera, these frames are put into the same queue. A cognitive engine will process the frame at the head

of the queue for the source it consumes frames from. This frame will be left at the head of the queue, so other cognitive engines that consume frames from the same source will also process it. This frame is only removed from the queue when a cognitive engine finishes processing it. The engine that finishes processing the frame at the head of the queue will be the first to get the next frame in the queue (if there is one).

This mechanism ensures that frames get consumed at the rate that the fastest cognitive engine can process them. If every cognitive engine took a frame from the queue without leaving it for other engines, a slower engine would process frames that the fastest engine does not. This method also ensures that cognitive engines do not get stale frames because they are slow. Cognitive engines always process the frame at the head of the queue. If one engine is slow, and the queue gets advanced several times while the slow engine is processing a single frame, the slow engine ignores the frames it missed. Once this engine becomes free, it will start processing the frame at the head of the queue at that time.

5.2.2 Components

Almost all of our applications use a single cognitive engine. Our server code runs workflows like this as a single Python program. A WebSocket server is run in the main process, and the cognitive engine is run in a separate process using Python’s multiprocessing module. Inter-process communication is done using the multiprocessing module’s Pipe function. For workloads that require multiple cognitive engines (such as the one depicted in Figure 8), the WebSocket server is run as a standalone Python program and each cognitive engine is run as a different Python program. The Python programs communicate with each other using ZeroMQ [14].

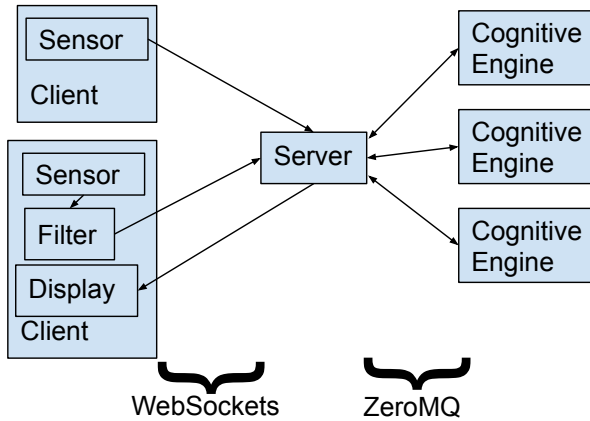


Figure 8: A Gabriel workflow with two clients and three cognitive engines.

We have developed client libraries for Python and Android. These include networking components that communicate with our server code using WebSockets. The libraries also contain functions to capture images with a camera and transmit the latest frame whenever a token is available. The Python library uses OpenCV [18] to capture images while the Android library uses CameraX [2]. Our Python code has been published to The Python Package Index (PyPI) [6, 7] and our Android code has been published to Maven Central [4].

Except for possible minor extensions, our current runtime platform will be sufficient to support everything proposed in this document. We therefore do not propose significant new work in this area.

6 Expected Timeline

All work on the runtime platform has been completed. The Stirling engine assistant has been completed. We have developed an initial version of the human escalation functionality. In addition, we have trained some models using synthetic images.

We are presently developing WCA applications for an Ikea Raksog cart and a model car. In addition, we are writing a technical report about our experiences with synthetic images.

We need to generate synthetic images using the CAD models that objects were manufactured from. We will then develop WCA applications that use these models. In addition, we will extend Ajalon to support fine-grained image classification in addition to object detection. Lastly, we will establish a process to improve our computer vision models using data that gets captured when a person completing a task requests help from a human expert.

I will defend my thesis in December 2022.

6.1 Completed Work

Time		Plan
2020 Summer	May – Aug	New Gabriel runtime components
2020 Fall	Sept – Dec	Implement escalation to human task expert
2021 Spring	Jan – May	WCA application for first ten steps of the Stirling engine task
2021 Summer	May – Aug	Full Stirling engine application Initial experiments with synthetic data

6.2 Remaining Work

Time		Plan
2021 Fall	Sept – Dec	WCA applications for Ikea cart and model car Improvements to Ajalon Model trained from synthetic data for small assembly
2022 Spring	Jan – May	Model trained from synthetic data for large assembly Improving model from data captured during call with human expert
2022 Summer	May Jun – Aug	Complete outstanding implementation work Dissertation writing
2022 Fall	Sept – Nov Dec	Dissertation writing Thesis defense

References

- [1] Zoom client sdks. <https://marketplace.zoom.us/docs/sdk/native-sdks/>. 3.2
- [2] Camerax overview. <https://developer.android.com/training/camerax>. 5.2.2
- [3] Dynamics 365 remote assist. <https://dynamics.microsoft.com/en-us/mixed-reality/remote-assist/>. 3.1
- [4] Gabriel android client. <https://repo1.maven.org/maven2/edu/cmu/cs/gabriel/>,. 5.2.2
- [5] The gabriel framework for wearable cognitive assistance using cloudlets. <https://github.com/cmusatyalab/gabriel>,. 5
- [6] Gabriel python client. <https://pypi.org/project/gabriel-client/>,. 5.2.2
- [7] Gabriel server. <https://pypi.org/project/gabriel-server/>,. 5.2.2
- [8] Lamp assistant. <https://github.com/cmusatyalab/gabriel-ikea>. 1, 2.1
- [9] Lego assistant. <https://github.com/cmusatyalab/gabriel-lego>. 1
- [10] Sandwich assistant. <https://github.com/cmusatyalab/gabriel-sandwich>. 1, 2.1
- [11] Wearable cognitive assistant for stirring engine. <https://github.com/cmusatyalab/gabriel-stirling-engine>,. 2.2.1
- [12] Stirling engine disassembly assistant - youtube. https://youtu.be/tU8jyDh_DGs,. 2.2.1
- [13] The websocket protocol. <https://tools.ietf.org/html/rfc6455>. 5.2.1
- [14] Zeromq. <https://zeromq.org/>. 5.2.2
- [15] Mario Aehnel and Bodo Urban. Follow-me: Smartwatch assistance on the shop floor. In Fiona Fui-Hoon Nah, editor, *HCI in Business*, pages 279–287, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07293-7. 1.2
- [16] Stavros Antifakos, Florian Michahelles, and Bernt Schiele. Proactive instructions for furniture assembly. In Gaetano Borriello and Lars Erik Holmquist, editors, *UbiComp 2002: Ubiquitous Computing*, pages 351–360, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45809-8. 1.2
- [17] Sebastian Bader and Mario Aehnel. Tracking assembly processes and providing assistance in smart factories. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence - Volume 1*, ICAART 2014, page 161–168, Setubal, PRT, 2014. SCITEPRESS - Science and Technology Publications, Lda. ISBN 9789897580154. doi: 10.5220/0004822701610168. URL <https://doi.org/10.5220/0004822701610168>. 1.2
- [18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 5.2.2
- [19] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiry-

- ong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, Daniel Siewiorek, and Mahadev Satyanarayanan. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350877. doi: 10.1145/3132211.3134458. URL <https://doi.org/10.1145/3132211.3134458>. 1
- [20] J. Deng, K. Li, M. Do, H. Su, and L. Fei-Fei. Construction and Analysis of a Large Scale Image Ontology. Vision Sciences Society, 2009. 2.1
- [21] Debidatta Dwibedi, Ishan Misra, and Martial Hebert. Cut, paste and learn: Surprisingly easy synthesis for instance detection. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1310–1319, 2017. doi: 10.1109/ICCV.2017.146. 4.1, 4.3
- [22] Gwendolyn Foo, Sami Kara, and Maurice Pagnucco. Screw detection for disassembly of electronic waste using reasoning and re-training of a deep learning model. *Procedia CIRP*, 98:666–671, 2021. ISSN 2212-8271. doi: <https://doi.org/10.1016/j.procir.2021.01.172>. URL <https://www.sciencedirect.com/science/article/pii/S2212827121002031>. The 28th CIRP Conference on Life Cycle Engineering, March 10 – 12, 2021, Jaipur, India. 2.1
- [23] C. Ailie Fraser, Tovi Grossman, and George Fitzmaurice. Webuild: Automatically distributing assembly tasks among collocated workers to improve coordination. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, page 1817–1830, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346559. URL <https://doi.org/10.1145/3025453.3026036>. 1.2
- [24] Timnit Gebru, Jonathan Krause, Yilun Wang, Duyun Chen, Jia Deng, and Li Fei-Fei. Fine-grained car detection for visual census estimation, 2017. 2.1, 2.2.1
- [25] Ankit Gupta, Dieter Fox, Brian Curless, and Michael Cohen. Duplotrack: A real-time system for authoring and guiding duplo block assembly. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, page 389–402, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315807. doi: 10.1145/2380116.2380167. URL <https://doi.org/10.1145/2380116.2380167>. 1.2
- [26] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. Synthetic data for text localisation in natural images. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2315–2324, 2016. doi: 10.1109/CVPR.2016.254. 4.1
- [27] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 68–81, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327930. doi: 10.1145/2594368.2594383. URL <https://doi.org/10.1145/2594368.2594383>. 1, 5
- [28] Stefan Hinterstoisser, Vincent Lepetit, Paul Wohlhart, and Kurt Konolige. On pre-trained image features and synthetic images for deep learning. In Laura Leal-Taixé and Stefan

- Roth, editors, *Computer Vision – ECCV 2018 Workshops*, pages 682–697, Cham, 2019. Springer International Publishing. ISBN 978-3-030-11009-3. 4.1
- [29] Stefan Hinterstoisser, Olivier Pauly, Hauke Heibel, Martina Marek, and Martin Bokeloh. An annotation saved is an annotation earned: Using fully synthetic training for object instance detection. *CoRR*, abs/1902.09967, 2019. URL <http://arxiv.org/abs/1902.09967>. 4.1
 - [30] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. The case for offload shaping. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile ’15, page 51–56, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333917. doi: 10.1145/2699343.2699351. URL <https://doi.org/10.1145/2699343.2699351>. 2.2.2
 - [31] Steven Johnson, Madeleine Gibson, and Bilge Mutlu. Handheld or handsfree? remote collaboration via lightweight head-mounted displays and handheld devices. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW ’15, page 1825–1836, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450329224. doi: 10.1145/2675133.2675176. URL <https://doi.org/10.1145/2675133.2675176>. 1.2
 - [32] K J Joseph, Salman Khan, Fahad Shahbaz Khan, and Vineeth N Balasubramanian. Towards open world object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2021)*, 2021. 3.2
 - [33] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013. 2.1, 4
 - [34] Benjamin Lafreniere, Tovi Grossman, Fraser Anderson, Justin Matejka, Heather Kerrick, Danil Nagy, Lauren Vasey, Evan Atherton, Nicholas Beirne, Marcelo H. Coelho, Nicholas Cote, Steven Li, Andy Nogueira, Long Nguyen, Tobias Schwinn, James Stoddart, David Thomasson, Ray Wang, Thomas White, David Benjamin, Maurice Conti, Achim Menges, and George Fitzmaurice. Crowdsourced fabrication. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST ’16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341899. doi: 10.1145/2984511.2984553. URL <https://doi.org/10.1145/2984511.2984553>. 1.2
 - [35] Peihua Li, Jiangtao Xie, Qilong Wang, and Zilin Gao. Towards faster training of global covariance pooling networks by iterative matrix square root normalization. In *IEEE Int. Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 2.1, 2.2.1
 - [36] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL <http://arxiv.org/abs/1405.0312>. 2.1
 - [37] Julien Miranda., Stanislas Larnier., Ariane Herbulot., and Michel Devy. Uav-based inspec-

tion of airplane exterior screws with computer vision. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 4: VISAPP*, pages 421–427. INSTICC, SciTePress, 2019. ISBN 978-989-758-354-4. doi: 10.5220/0007571304210427. 2.1

- [38] Gabriela Motroc. “a machine learning model is only as good as the data it is fed”. <https://jaxenter.com/apache-spark-machine-learning-interview-143122.html>. 3
- [39] Truong An Pham, Junjue Wang, Roger Iyengar, Yu Xiao, Padmanabhan Pillai, Roberta Klatzky, and Mahadev Satyanarayanan. Ajalon: Simplifying the authoring of wearable cognitive assistants. *Software: Practice and Experience*, 51(8):1773–1797, 2021. doi: <https://doi.org/10.1002/spe.2987>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2987>. 1
- [40] Param S. Rajpura, Ravi S. Hegde, and Hristo Bojinov. Object detection using deep cnns trained on synthetic images. *CoRR*, abs/1706.06782, 2017. URL <http://arxiv.org/abs/1706.06782>. 4.1
- [41] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL <https://proceedings.neurips.cc/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf>. 1, 2.1
- [42] Mahadev Satyanarayanan. The Emergence of Edge Computing. *IEEE Computer*, 50(1), 2017. 1
- [43] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *6th International Conference on Mobile Computing, Applications and Services*, pages 1–9, 2014. doi: 10.4108/icst.mobicaase.2014.257757. 1
- [44] Herbert A. Simon. *The Architecture of Complexity*, pages 457–476. Springer US, Boston, MA, 1991. ISBN 978-1-4899-0718-9. doi: 10.1007/978-1-4899-0718-9_31. URL https://doi.org/10.1007/978-1-4899-0718-9_31. 2.1, 2.2.4
- [45] Arjun Singh, James Sha, Karthik S. Narayan, Tudor Achim, and Pieter Abbeel. Bigbird: A large-scale 3d database of object instances. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 509–516, 2014. doi: 10.1109/ICRA.2014.6906903. 4.3
- [46] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1082–10828, 2018. doi: 10.1109/CVPRW.2018.00143. 4.1
- [47] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects.

CoRR, abs/1809.10790, 2018. URL <http://arxiv.org/abs/1809.10790>. 4.1

- [48] A. Vedaldi, S. Mahendran, S. Tsogkas, S. Maji, B. Girshick, J. Kannala, E. Rahtu, I. Kokkinos, M. B. Blaschko, D. Weiss, B. Taskar, K. Simonyan, N. Saphra, and S. Mohamed. Understanding objects in detail with fine-grained attributes. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014. 4
- [49] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. The Caltech-UCSD Birds-200-2011 Dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011. 4
- [50] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC '19*, page 152–165, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367332. doi: 10.1145/3318216.3363308. URL <https://doi.org/10.1145/3318216.3363308>. 1
- [51] Junhui Wu, Yun Ye, Yu Chen, and Zhi Weng. Spot the difference by object detection, 2018. 2.1