

# Scaling Up Wearable Cognitive Assistance for Assembly Tasks

Roger Iyengar

CMU-CS-23-112

April 2023

Computer Science Department  
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 © 2023 Roger Iyengar

This research was sponsored by Autodesk, the National Science Foundation under award numbers 1518865 and 2106862, the NSF Graduate Research Fellowship under Grants DGE1252522 and DGE1745016, and the Defense Advanced Research Projects Agency under award number HR001117C0051.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

**Keywords:** Wearable Cognitive Assistance, Wearable Computing, Edge Computing, Split Computing, Computer Vision, Synthetic Data

*For Rajan Laddu, Sujeath Pareddy, and Joel Reidenberg.  
May your memories be a blessing.*



## **Abstract**

Wearable Cognitive Assistance (WCA) applications run on wearable mobile devices, to provide guidance for real world tasks. Physical assembly tasks have been a significant focus of research on WCA. We introduce new techniques to support the development of WCA applications for more complex assembly tasks than previous techniques supported. In addition, our work reduces the load on developers creating WCA applications by eliminating the need to collect and label real training images. We accomplish this by training computer vision models on synthetically generated images. This dissertation investigates escalation to human experts in cases when a user is not satisfied with the automated guidance from an application. Lastly, we develop a new version of a software framework for WCA applications, and evaluate ways in which WCA applications can benefit from running computations directly on mobile devices.



## Acknowledgments

I am deeply indebted to my advisor Satya. Satya has taught me a lot about approaching difficult problems, conducting research, selling ideas, and communicating results. In addition, he has been a strong advocate for me and he has done a lot to help ensure my successful completion of the program. I also owe a great deal to the members of my thesis committee: Martial Hebert, Roberta Klatzky, and Padmanabhan Pillai. They have provided valuable guidance on this research, and helped improve the quality of this document.

Satya's research group is an extremely collaborative environment, where people are always willing to help each other out. Thank you to Mihir Bala, Jim Blakley, Jason Choi, Tom Eiszler, Ziqiang Feng, Shilpa George, Jan Harkes, Chanh Nguyen, Eric Sturzinger, and Junjue Wang for making this such a productive and pleasant place to work. Additional thanks are due to Jan and Tom for managing and supporting all of our group's servers.

I was fortunate enough to mentor Qifei Dong, Max Krieger, Di Wang, and Emily Zhang; all of whom contributed significantly to this work. I also had the privilege of being mentored by Professors Brendan Juba, Joe Near, Norman Sadeh, Dawn Song, and Sebastian Zimmeck prior to my work with Satya. I owe a debt of gratitude to my other research collaborators: Steven M. Bellovin, Sushain Cherivirala, Lorie Cranor, Yuanyuan Feng, Hana Habib, Margaret Hagan, Bin Liu, Vinayshekhar Bannihatti Kumar, Namita Nisal, Truong An Pham, Joel Reidenberg, Michel Roy, Florian Schaub, Peter Story, Om Thakkar, Abhradeep Thakurta, Haithem Turki, Lun Wang, Ziqi Wang, Yu Xiao, Shomir Wilson, and Lieyong Zou. On top of this, I have benefited substantially from anonymous reviews of my papers. Additional thanks are due to the many teachers I have had throughout the course of my life.

Thank you to Wael Darwich for your development work on A3D, for your work generating synthetic images, and for helping me get A3D set up to generate my own synthetic images. Haley Edie was instrumental in coordinating our collaboration with Autodesk, and getting me access to Autodesk resources. David Lovell ensured that the phone sanitizer parts were manufactured and shipped to us, and he captured some of our early test data.

Rod Heiple handled the Autodesk collaboration on the CMU side. Ryan Bates 3D printed parts for the toy plane. Deb Cavlovich, Catherine Copetas, Sara Golem-biewski, Connie Herold, Chase Klingensmith, Nichole Merritt, Colleen Mollenauer, and Linda Moreci provided fantastic administrative support and helped me navigate several bureaucratic processes.

Writing software in the 21st century is standing on the shoulders of giants. This work would not have been possible without the developers of GNU, Linux, LLVM, Cuda, PyTorch, TensorFlow, Android, CPython, NumPy, SciPy, Matplotlib, OpenCV, CVAT, fish shell, Emacs, pubd, IntelliJ, Docker, AIOHTTP, Jupyter Notebook, ZeroMQ, and countless other software projects.

Lastly, thank you to all of my family and friends, especially my parents Arun and Louise Iyengar. I would not be here without your love and support.





# Contents

- 1 Introduction 1**
  - 1.1 Thesis Statement . . . . . 2
  - 1.2 Potential Impact . . . . . 2
  - 1.3 Novelty . . . . . 4
  - 1.4 Roadmap . . . . . 4
  
- 2 Background 5**
  - 2.1 Aids for Assembly Tasks . . . . . 5
  - 2.2 Wearable Cognitive Assistance . . . . . 7
  - 2.3 Computer Vision . . . . . 8
  - 2.4 Synthetic Training Data . . . . . 8
  - 2.5 Hierarchical Decomposition . . . . . 9
  - 2.6 Development Toolchain . . . . . 9
  
- 3 Detecting Completed Steps of Complex Tasks 13**
  - 3.1 Hierarchical Decomposition . . . . . 14
  - 3.2 Two Stage Process . . . . . 16
    - 3.2.1 Re-Using Labels . . . . . 17
    - 3.2.2 Training . . . . . 17
    - 3.2.3 Error Correction . . . . . 17
    - 3.2.4 Development Tools . . . . . 18
  - 3.3 Our Applications . . . . . 18
    - 3.3.1 Stirling Engine . . . . . 19
    - 3.3.2 Ikea Cart . . . . . 20
    - 3.3.3 Toy Car . . . . . 20
  - 3.4 Implementation Details . . . . . 25
  - 3.5 Guidelines for WCA Developers . . . . . 26
    - 3.5.1 Subassemblies . . . . . 26
    - 3.5.2 Training Data . . . . . 27
  
- 4 Accelerate WCA development with synthetic training images 29**
  - 4.1 Meccano Erector Kit . . . . . 29
    - 4.1.1 Generating Data . . . . . 30
    - 4.1.2 Results . . . . . 36

4.2	Toy Plane . . . . .	37
4.3	Phone Sanitizer . . . . .	39
4.4	Discussion . . . . .	43
<b>5</b>	<b>Escalation to Human Experts</b>	<b>45</b>
5.1	Experts Without Automation . . . . .	46
5.2	Calls With Human Experts . . . . .	46
5.3	Simulating Call Centers . . . . .	49
5.3.1	A Simple Model . . . . .	49
5.3.2	Lognormal Service Times . . . . .	52
5.3.3	Simulating All Steps . . . . .	54
5.4	Extending to Real Call Centers . . . . .	59
5.5	Exploring Parameter Space . . . . .	59
5.5.1	Varying the Proportion of Feasible Steps . . . . .	59
5.5.2	Varying Patience Length . . . . .	60
5.5.3	Varying Length of Feasible Steps . . . . .	60
5.6	Limiting the Number of Active Users . . . . .	62
5.7	Summary . . . . .	63
<b>6</b>	<b>Device and Cloudlet Implementation</b>	<b>65</b>
6.1	Software Framework . . . . .	65
6.1.1	Motivation . . . . .	65
6.1.2	Key Abstractions . . . . .	66
6.2	Leveraging Mobile Device Hardware . . . . .	69
6.2.1	Accuracy Comparisons . . . . .	69
6.2.2	On-device WCA . . . . .	72
6.2.3	Split Computing . . . . .	74
6.2.4	Thin Clients . . . . .	79
6.3	Summary . . . . .	81
<b>7</b>	<b>Conclusion and Future Work</b>	<b>83</b>
7.1	Contributions . . . . .	83
7.2	Future Work . . . . .	84
7.2.1	Subassembly Identification . . . . .	84
7.2.2	Detecting Environmental Issues . . . . .	84
7.2.3	Computer Vision Techniques . . . . .	84
7.2.4	Textures for 3D Models . . . . .	85
7.2.5	Device and Cloudlet Implementations . . . . .	85
7.2.6	Development Tools . . . . .	85
7.2.7	Multi-Modal Sensing . . . . .	86
	<b>Bibliography</b>	<b>87</b>

# List of Figures

- 1.1 A comparison of gross and subtle differences in task steps . . . . . 3
  
- 3.1 A stirling engine with two sub-assemblies highlighted . . . . . 15
- 3.2 A model motorcycle from a Meccano Erector kit . . . . . 15
- 3.3 The architecture of our WCA applications . . . . . 16
- 3.4 The issue with identical consecutive steps . . . . . 18
- 3.5 The steps detected by our Stirling Engine WCA application . . . . . 19
- 3.6 Four states from our WCA application for a Stirling engine . . . . . 20
- 3.7 The fully assembled utility cart . . . . . 21
- 3.8 The steps detected by our Ikea WCA application . . . . . 22
- 3.9 The fully assembled model car . . . . . 23
- 3.10 The steps detected by our Toy Car WCA application . . . . . 24
- 3.11 Images of the toy car and Ikea cart kits, before and after being cropped . . . . . 25
- 3.12 Images with identical and different perceptual hash values . . . . . 26
  
- 4.1 Examples of synthetically generated images . . . . . 30
- 4.2 The fully assembled model bike . . . . . 31
- 4.3 A synthetic image showing part of the bike model . . . . . 32
- 4.4 Bounding boxes with and without padding . . . . . 33
- 4.5 Our first attempt at making our synthetic images look more realistic . . . . . 34
- 4.6 Our model incorrectly detected a line in the floor as an object of interest . . . . . 35
- 4.7 Synthetically generated images from our final set . . . . . 35
- 4.8 Examples Images from Adobe Stock that we used as background textures . . . . . 36
- 4.9 The fully assembled model plane kit . . . . . 38
- 4.10 CAD models for the individual model plane parts . . . . . 38
- 4.11 CAD models for the model plane assembly steps . . . . . 38
- 4.12 Synthetically generated training images for the toy plane kit . . . . . 39
- 4.13 The fully assembled Phone Sanitizer . . . . . 40
- 4.14 Synthetic images of the phone sanitizer, from the Unity Perception Package . . . . . 41
- 4.15 Synthetic images of the phone sanitizer, from Autodesk A3D . . . . . 42
- 4.16 Images generated using A3D with simple object textures . . . . . 43
  
- 5.1 The design space of remote expert assistance systems for assembly tasks . . . . . 47
- 5.2 The workflow followed to request help from a human expert . . . . . 47
- 5.3 The components of a WCA application with human assistance . . . . . 48

5.4	A screenshot of the web application used by the human task expert . . . . .	48
5.5	The PDFs for the distributions that Simulation 1 samples from . . . . .	50
5.6	The waiting times resulting from Simulation 1 and Formula 5.1 . . . . .	51
5.7	The PDFs for the distributions that Simulation 2 samples from . . . . .	52
5.8	The waiting times resulting from Simulation 2 and Formula 5.4 . . . . .	53
5.9	The PDFs for the distributions that Simulation 3 samples from . . . . .	55
5.10	Inter-entrance time samples from Simulation 3 . . . . .	56
5.11	Inter-arrival times for help calls that resulted from Simulation 3 . . . . .	56
5.12	Waiting times from Simulation 3 and Formula 5.4 . . . . .	58
5.13	Waiting times from Simulation 3 plotted against times from Formula 5.4 . . . . .	58
5.14	Average wait times resulting from changing the proportion of feasible steps . . . . .	60
5.15	Average wait times resulting from changing users' simulated patience . . . . .	61
5.16	Average wait times resulting from changing the average step length . . . . .	61
5.17	Average wait times resulting from Simulation 4 . . . . .	62
5.18	The number of users that were serviced under certain wait time thresholds . . . . .	63
6.1	Two Gabriel clients that produce frames from multiple sensors . . . . .	67
6.2	Two cognitive engines consuming frames from the same queue . . . . .	68
6.3	A Gabriel workflow with two clients and three cognitive engines . . . . .	69
6.4	Images showing steps from the Stirling and Meccano tasks . . . . .	71
6.5	Power consumption for headsets running EfficientDet inference in a loop . . . . .	73
6.6	On-device, split computing, and thin client implementations of WCA . . . . .	76
6.7	The network connecting mobile devices to the cloudlet . . . . .	78

# List of Tables

- 2.1 Existing systems that helped users with physical assembly tasks . . . . . 6
- 2.2 Example Wearable Cognitive Assistance Applications . . . . . 10
- 2.3 Resources used by example WCA applications . . . . . 11
  
- 4.1 Classification results for model pairs trained on data for the Meccano kit . . . . . 37
- 4.2 Classification results for model pairs trained on data for the toy plane . . . . . 39
- 4.3 Classification results for model pairs trained on data for the phone sanitizer . . . . . 41
  
- 5.1 The variables used to compute expected wait time for users in an M/M/N queue . . . . . 50
- 5.2 Parameter values for the distributions that Simulation 1 samples from . . . . . 51
- 5.3 Parameter values for the distributions that Simulation 2 samples from . . . . . 53
- 5.4 Parameter values for the distributions that Simulation 3 samples from . . . . . 57
  
- 6.1 A summary of the data used for the experiments in Chapter 6 . . . . . 70
- 6.2 Classification accuracy for standalone DNN models . . . . . 70
- 6.3 Classification accuracy for pipelines . . . . . 71
- 6.4 The smart glasses that we profiled our applications on . . . . . 72
- 6.5 Inference time for one frame, in milliseconds . . . . . 72
- 6.6 The largest pipeline that meets tight and loose latency bounds . . . . . 73
- 6.7 Average power consumption, in Watts . . . . . 74
- 6.8 The percentage increase of power consumption, above the baseline . . . . . 74
- 6.9 The bandwidth saved by transmitting cropped images . . . . . 77
- 6.10 Single-frame inference time of split computing pipelines . . . . . 78
- 6.11 The largest split pipelines that meet tight and loose latency bounds . . . . . 78
- 6.12 Accuracy of pipelines from Tables 6.10 and 6.11 . . . . . 78
- 6.13 Average power consumption of mobile devices running DNN pipelines . . . . . 79
- 6.14 The percentage increase of power consumption, above the baseline . . . . . 79
- 6.15 Classification accuracy for pipelines . . . . . 80
- 6.16 Single-frame inference time for the thin client . . . . . 80
- 6.17 Average power consumption for thin clients, in Watts . . . . . 81



# Chapter 1

## Introduction

Wearable Cognitive Assistance (WCA) applications provide guidance to users for a specific task. This task could range from assembling a physical object, remembering people’s names, exercising, or playing a game or sport. These applications utilize mobile devices, such as smart glasses or a smartphone, to capture data and interact with the user. WCA applications process captured data in order to determine the physical state of the task, and then provide assistance based on this physical state.

Table 2.2 lists some examples of WCA applications that have been developed prior to the work in this dissertation. All of these applications determine a task’s physical state based on images from an RGB camera. This camera 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. Further, computation must be offloaded to a server with close network proximity to the mobile device that is capturing the data [79]. We will henceforth refer to this server as a cloudlet.

This work 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. The application accomplishes this by processing frames from an RGB camera. Prior to the work reported here, applications had been developed for a lego kit [7], a lamp [6], and a toy sandwich [9]. 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 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 in which the model performs poorly. Creating training sets is thus a significant barrier to developing WCA applications that support large numbers of steps and large numbers of parts.

Assembly tasks might involve parts that are much smaller than the full object being assembled. For example, one step might require the user to insert a screw into a large metal piece. The application needs to be able to detect when steps involving small screws have been completed, as well as steps involving larger parts. In addition, there are many possible ways that a user can complete a task step incorrectly. In fact, the number of possible errors that a person can make while completing a task is significantly larger than the number of steps that are required to complete the task. These challenges lead to my thesis statement.

## 1.1 Thesis Statement

**Scaling up WCA to complex assembly tasks 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 Potential Impact

The following issues currently make WCA applications impractical for real world assembly tasks:

- Current techniques do not support tasks with more than roughly ten steps, and the steps must have gross visible differences from each other. For example, a WCA application developed with these techniques cannot detect the presence or absence of a single screw. Figure 1.1 shows an example of the gross differences that current techniques support, and the subtle differences that require the techniques introduced in this work.
- A developer must collect and label training data for each error state that the application detects. However, there are exponentially more error states than correct states for any given task. This makes it infeasible to create a WCA application that is capable of detecting all possible error states for a task.
- The computer vision models that we use for WCA applications require thousands of images depicting each state. All of these images must be labeled with a bounding box. Collecting and labeling these images requires an immense manual effort.



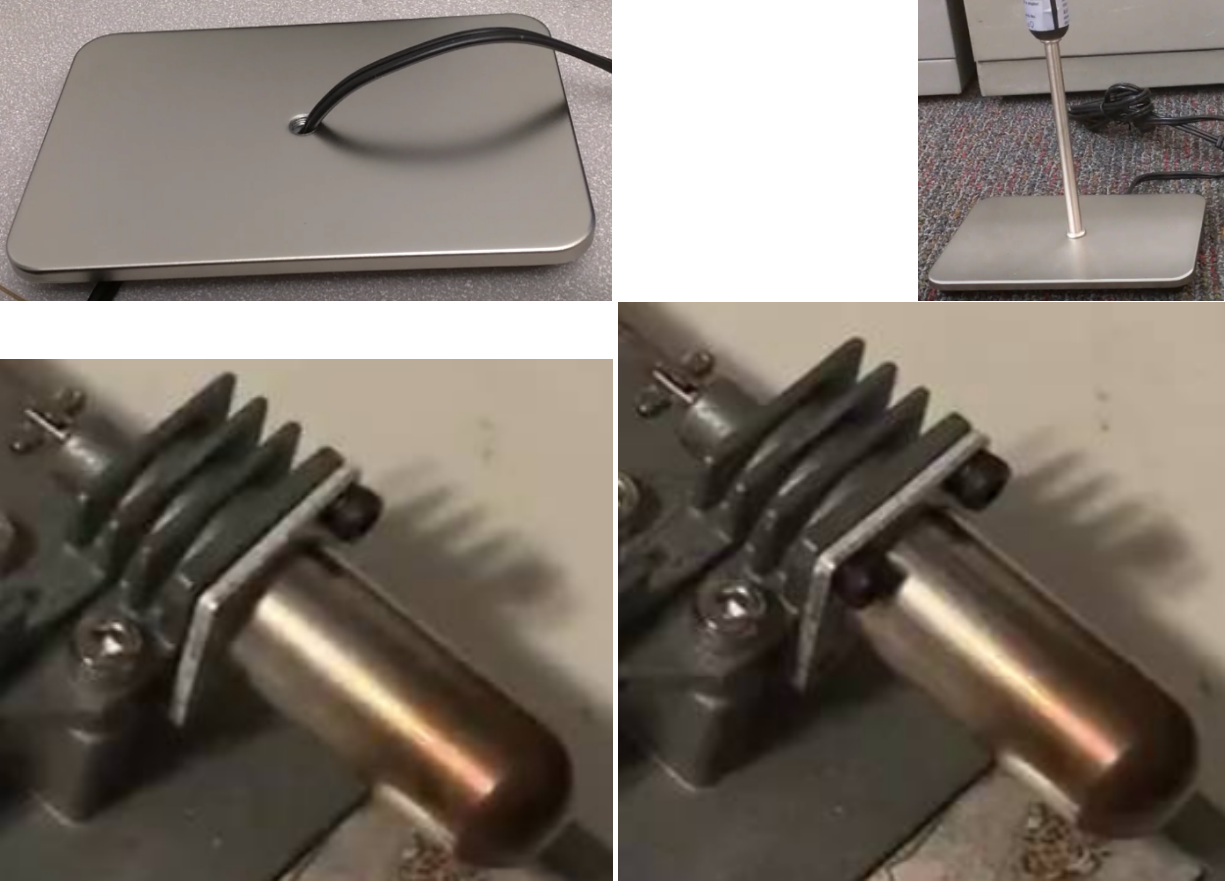


Figure 1.1: A comparison of gross and subtle differences in task steps. The top two images illustrate a gross difference that current WCA techniques can detect. The bottom two images show a subtle difference that requires the new techniques from this work to support.

This work addresses all three of these issues. We present new techniques that enable the development of WCA apps with steps that look almost identical. The changes between these steps are subtle, such as the addition of a single screw. Our techniques also enable the development of tasks with a larger number of steps.

Existing WCA applications for assembly tasks have no way of handling user errors that they were not specifically developed to support. If a user makes such a mistake, the application will either provide the user with an instruction for a different step, that it was built to recognize, or it might not provide any guidance at all. In these cases, we allow the user to start a video call with a human who is an expert on the task that the user is trying to complete. We call this feature *human escalation*.

Lastly, reducing the time it takes for developers to build WCA applications for assembly tasks is imperative for motivating a critical mass of developers to build them. Training computer vision models using synthetic data entirely removes the need to manually capture images of each task step and label these images with bounding boxes.

## 1.3 Novelty

Existing WCA applications use techniques that cannot support assembly tasks with large numbers of steps or subtle differences between steps. An example of a subtle difference is the insertion or removal of a single screw (as depicted in Figure 1.1). The lego application developed by Chen et al. [30] guided users to place colored blocks on a 3D grid. This application did not use any machine learning (ML) models, and its techniques are tightly coupled to this specific task. The Sandwich application developed by Chen et al. [30] determined the task step that was shown using a single Faster R-CNN object detector. As we describe in §6.2.1, a single Faster R-CNN object detector is not sufficient for more realistic tasks.

The combinatorial explosion of possible error states has not been addressed in any prior work on WCA applications. The sandwich application from Chen et al. [30] was built to detect one specific error. But none of our prior work has attempted to address the wide range of possible mistakes that a well-intentioned user might make when completing an assembly task. Escalation to a human expert is the first solution that has been proposed.

Training ML models using synthetic images has long been an area of active research. However, this work represents the first attempt to do so in the context of WCA. We offer results from training models for real WCA applications, using synthetic data.

## 1.4 Roadmap

This dissertation validates the thesis as follows:

- Chapter 2 discusses prior work on WCA, other work on aids for assembly tasks, and the computer vision research that we leverage in our work.
- Chapter 3 describes how we detect when steps of real world tasks have been completed. It also presents the three WCA applications that we developed using our techniques.
- Chapter 4 presents and evaluates our efforts to train DNNs for WCA applications using synthetic training images.
- Chapter 5 introduces our system for error handling with human task experts. The chapter also presents Monte Carlo simulations that call center operators can use to determine the number of human experts that are required to support a set of WCA application users.
- Chapter 6 describes the software framework that we built to support Gabriel applications, and it explores how mobile device hardware can be used by WCA applications.
- Chapter 7 concludes the dissertation and offers suggestions for future work on WCA applications.

# Chapter 2

## Background

This work builds upon prior work on WCA applications, and it leverages existing work in computer vision. This chapter provides an overview of this work in addition to a comparison between WCA and other computerized aids for assembly tasks.

### 2.1 Aids for Assembly Tasks

We consider three types of people who are involved with WCA applications. The first are *users*, who are completing a task and receiving guidance from the application. The next are *experts*, who are familiar with the tasks, and the mistakes that a user might make. Some systems (including ours) allow experts to help users with a task, over a video call. The last type of people are *developers* who create a WCA application for a specific task, but developers are not involved at the point that a user is completing the task.

A large body of work has been done by other researchers 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. In contrast to some previous work, 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. In addition, our system does not require the continuous attention of a task expert to observe a user for the entire duration of that task. The application starts out providing automated guidance, and only starts a call with a task expert when the user requests help from a human. 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. [41] 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. Their system has no way to automatically detect when a step is completed, and it does not handle user errors. 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. [22] developed a system that determines when steps for assembling an Ikea wardrobe have been completed, using sensors such as gyroscopes,

Prior Work	Detection of Step Completion	Other Attributes
Fraser et al. [41]	User presses a button	Multiple users working on task together.
Antifakos et al. [22]	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. [46]	Automatic, using Microsoft Kinect	Builds a full virtual model of what the user has constructed. Only supports large colorful duplo blocks.
Aehnel and Urban [21]	Automatic, using RFIDs and infrared light barriers	Requires sensors to be present in working environment.
Lafreniere et al. [60]	Manually from user	The final assembled object was massive. Many parts were identical.
Johnson et al. [54]	Remote task expert	Compared tablet with Google Glass.

Table 2.1: Existing systems that helped users with physical assembly tasks

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. [46] 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 simpler shapes than the objects that our tasks use. Aehnel and Urban [21] 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 [24]. Our system does not require the presence of such sensors.

Lafreniere et al. [60]’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. [54] 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 2.1 provides a brief summary of all of the existing systems described in this section.

## 2.2 Wearable Cognitive Assistance

A WCA application provides just-in-time guidance and error detection for a user who is performing an unfamiliar task. Prompt error detection is also valuable for a user who is performing familiar tasks, since human errors cannot be completely avoided, especially when the user is tired or stressed. Informally, WCA is like having “an angel on your shoulder.” [48] It broadens, the metaphor of GPS navigation tools that provide real-time step-by-step guidance, with prompt error detection and correction.

This work builds on top of past research on WCA applications. Ha et al. [48] introduced the first version of a programming framework called Gabriel. This framework includes networking and runtime components for WCA applications. Chen et al. [30] 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. [87] examined how to reduce the load imposed on a cloudlet by a single WCA user, thereby allowing many more users to share single cloudlet. Pham et al. [72] developed a toolchain that allows people to develop WCA applications without writing any code.

Table 2.2 lists some examples of WCA applications that have developed in prior work. In total, those researchers have developed over 15 WCA applications. This work focuses on applications that that help users assemble physical objects. Chen et al. [30] developed such applications for an IKEA lamp, a Lego kit, and a toy sandwich. We developed applications for an IKEA cart, a model car, a smartphone sanitizer, a model plane, a Meccano bike it, and a Stirling engine.

These applications are a compelling use case for edge computing. They first capture images using the camera on a mobile device, such as a smartphone or head-mounted wearable device. The images are then sent to a cloudlet for processing, using the Gabriel platform [48]. The computational limitations of lightweight mobile devices that have acceptable battery life prevent applications from processing images using the devices’ own hardware [78]. Table 2.3 lists the resource consumption and end-to-end latency bounds of five offloading-based WCA applications. It shows that WCA applications are simultaneously compute-intensive, bandwidth-hungry, and latency-sensitive.

Our work extends this body of research in the following ways:

- Utilizing new computer vision techniques to support more realistic assembly tasks.
- Adding live call support to WCA applications, so human experts can help users correct errors they make with tasks.
- Training computer vision models using synthetic training images.
- Developing a new software framework for WCA applications, that allows multiple users to share a single instance of an application running on a cloudlet.
- Exploring how well DNNs running on mobile device hardware can support WCA applications.

## 2.3 Computer Vision

Many of the aids for assembly tasks described in Section 2.1 require the user to press a button in order to indicate that a step has been completed. However, our applications determine when steps have been completed automatically, using computer vision.

This work’s contribution is in how computer vision (CV) is being applied, rather than developing any new CV algorithms. This section highlights the CV research that we leverage in our work.

We follow the lead of Gebru et al. [42], who 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 leverage their approach to train models on our own new data using 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 [33] 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 [62] is an object detection dataset with 80 classes. Faster R-CNN [75] is an object detector that is used in our lamp [6] and toy sandwich [9] 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 [58] is more fine-grained, with images of cars that are labeled with the year, make, and model. The fine-grained dataset requires classifying cars based on small intricate details, while classifying cars into the three coarse-grained categories is simpler. This increase in complexity is similar to the increase in complexity between the work of Chen et al. [30] and the work presented in this dissertation. The Fast MPN-COV [61] image classifier performs well on several fine-grained classification datasets.

Our work is not the first to use object detection models for tasks beyond detecting the objects in an image and providing their coordinates. Object detectors have been used for finding screws in pictures of aircraft parts [66] and consumer electronics [40]. Wu et al. [90] found differences between book covers using a modified version of Faster R-CNN.

## 2.4 Synthetic Training Data

Training CV models in order to develop a WCA application for a specific task requires thousands of labeled images. Collecting and labeling this data requires a substantial effort, and it must be repeated for each new WCA application. Training models with synthetic data would eliminate or reduce the need for WCA application developers to collect and label real images. Reducing the amount of time it takes developers to create WCA applications helps make WCA applications practical for real world tasks.

The idea of avoiding manual labeling has a long and rich history. Hinterstoisser et al. [52] 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 [85, 47], or used real background images [50, 73, 84]. Dwibedi et al. [34] avoided rendering 3D graphics altogether by cropping objects from photographs, and pasting these crops into other photographs. Their models trained on synthetic images performed worse than their models trained on real images. However, they also trained models using a mix of real and synthetic images, and these performed better than models trained on real or synthetic data alone.

## 2.5 Hierarchical Decomposition

Simon [81] 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 smaller sub-assemblies. 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.

Epshtein and Ullman [35] utilized hierarchies of visual features to recognize objects. Gong et al. [43] automatically identified subassemblies of assembled objects based on CAD files. They utilize features such as the number of other parts that a certain part touches, or the amount of surface area that two touching parts share. They considered subassemblies from the perspective of assembly sequence planning, rather than detecting completed task steps using computer vision. Separating complex tasks into subassemblies for WCA applications requires considering how easy it is for a user to complete a task, and how easy it is for computer vision models to detect when steps have been completed.

## 2.6 Development Toolchain

Pham et al. [72] created a set of tools called Ajalon, that developers can use to create WCA applications for assembly tasks. They utilized Intel's Computer Vision Annotation Tool (CVAT) [1] and developed Open Workflow Editor [15], a state machine editor. The authors ran a user study with developers to show that their toolchain is effective.



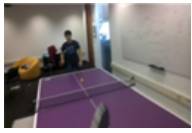







App Name	Example Input Video Frame	Description	Symbolic Representation	Example Guidance
<b>Pool</b>		Helps a novice pool player aim correctly. Gives continuous visual feedback (left arrow, right arrow, or thumbs up) as the user turns his cue stick. The symbolic representation describes the positions of the balls, target pocket, and the top and bottom of cue stick.	<Pocket, object ball, cue ball, cue top, cue bottom>	
<b>Ping-pong</b>		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.	<InRally, ball position, opponent position>	“Left!”
<b>Work-out</b>		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”
<b>Face</b>		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”
<b>Lego</b>		Guides a user in assembling 2D Lego models. The symbolic representation is a matrix representing color for each brick.	[[0, 2, 1, 1], [0, 2, 1, 6], [2, 2, 2, 2]]	“Put a 1x3 green piece on top”
<b>Draw</b>		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.		
<b>Sandwich</b>		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. [75]	“Lettuce on top of ham and bread”	“Put a piece of bread on the lettuce”

Table 2.2: 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 [77])



	Pool	Ping-pong	Face	Lego	Sandwich
Cloudlet CPU load(%)	72.10	45.40	75.60	52.20	85.10
End-to-end latency bounds (tight-loose, ms)*	95-105	150-230	370-1000	600-2700	
Video streaming bandwidth requirement (Average / Peak, Mbps)	480p: 3.6 / 7.0 720p: 6.8 / 9.9 1080p: 8.1 / 12.7				

Table 2.3: Resources used by example WCA applications. The implementations of the application servers [30] were tested on a laptop (with an Intel<sup>®</sup> Core<sup>™</sup> i7-8500Y processor and 8GB RAM), running the frontend on an Android phone, using 480p, 720p, and 1080p video resolutions. (\*) End-to-end latency includes both the round trip time (RTT) from the Android phone to the cloudlet and compute time in the cloudlet. Tight and loose bounds are adopted from Chen, et al, [30] where they are defined: “The tight bound represents an ideal target, below which the user is insensitive to improvements. Above the loose bound, the user becomes aware of slowness, and user experience and performance is significantly impacted.”



## Chapter 3

# Detecting Completed Steps of Complex Tasks

Many of the aids for assembly tasks described in Section 2.1 require the user to press a button in order to indicate that a step has been completed. This section examines how applications can use computer vision to automatically determine when steps have been completed. This frees users from having to use their hands to press a button on a wearable device after they have completed each step of a task.

A user wears a mobile device with a camera (such as a Google Glass). The mobile device 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 in order to give the camera a certain view. The application must determine when a step is completed based on images from the camera. It is important to avoid false positives, which are instances where the application determines that a step has been completed before it actually has been. False positives cause the application to give the user a new instruction before the previous step gets completed, which results in a poor user experience. When a false negative occurs, the application will not give the user a new instruction after they have completed a step. However, false negatives can typically be corrected by slightly rotating the assembly, which will give the camera a different view of the object and thus get the classifier to assign a correct label. False negatives result in a suboptimal user experience, but they are less disorienting than false positives.

Developers must train computer vision models to recognize when each step of the task has been completed. They must also train the models for each error state that they want the application to be able to recognize. Unfortunately, there is a combinatorial explosion in the number of possible error states, which we address in Chapter 5.

This chapter presents the approaches we have developed to detect the step of a physical task that is shown in an image, in the context of WCA. We wanted to develop something that takes an image captured by a headset camera, and determines the step of the task that is shown in that image. If a user has made a mistake that the application was developed to recognize, we want the application to be able to recognize this as well.

## 3.1 Hierarchical Decomposition

Inspired by Simon’s argument that all complex systems are made up of smaller systems [81], we argue that any object that is assembled from more than ten parts can be decomposed into sub-assemblies. The CAD programs Autodesk Inventor and Intergraph Smart 3D both represent assemblies as a hierarchy [23, 31]. Epshtein and Ullman [35] utilized hierarchies of visual features to recognize objects. Gong et al. [43] automatically identified subassemblies of assembled objects based on CAD files. They utilize features such as the number of other parts that a certain part touches, or the amount of surface area that two touching parts share.

The lower bound of ten parts comes from our firsthand experience developing WCA applications. We have successfully developed applications for objects assembled out of ten or fewer parts without splitting the task into subassemblies. However, for objects assembled using more than ten parts, splitting the task into subassemblies has proven to be a useful technique.

This technique 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 will 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. 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. Hierarchical Decomposition can be used to split a task that requires hundreds of steps, or even thousands of steps, into sub-assemblies that can be assembled in 10 steps or fewer. Our initial WCA applications were all for 10 step tasks. It thus made sense to develop WCA applications for longer tasks as a series of tasks that required 10 or fewer steps.

Figure 3.1 shows two of the sub-assemblies of a Stirling engine. Our application uses a different pair of computer vision models for each sub-assembly. The code selects the correct pair based on the current step that the user is working on. Each step involves attaching a piece to 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. The optimal number of steps for a sub-assembly may vary based on the task. However, limiting the number of steps to 10 worked well for all of the applications that we have developed.

Figure 3.2 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 requires an additional set of steps at the end, to connect the sub-assemblies together.

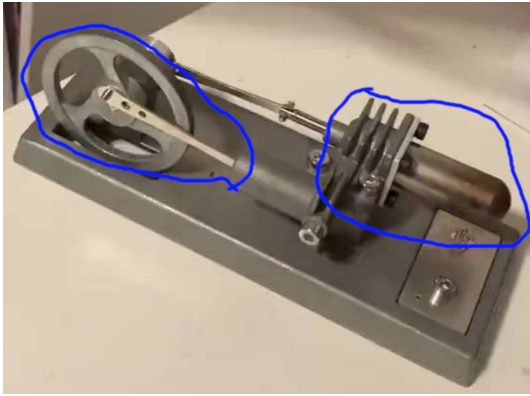


Figure 3.1: A stirling engine with two sub-assemblies highlighted



(a) The fully-assembled model



(b) Sub-assemblies

Figure 3.2: A model motorcycle from a Meccano Erector kit. The sub-assemblies are first assembled from smaller components. Next, the user combines the sub-assemblies together into the fully-assembled motorcycle. The application uses a fine-grained image classifier and an object detector for the final steps of putting the 3 sub-assemblies together. In these final steps, each of the 3 sub-assemblies is treated like a part that the user is attaching to the final object being assembled.

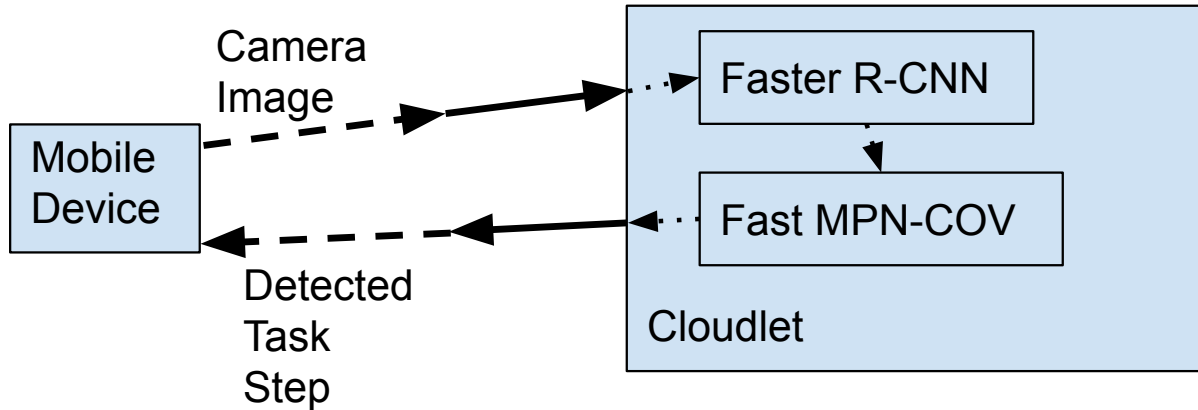


Figure 3.3: The architecture of our WCA applications. The dashed lines represent a Wi-Fi connection. The solid lines represent a connection over Gigabit Lan. The dotted lines represent data transmission between components on a single cloudlet.

## 3.2 Two Stage Process

An assembly task involves attaching components to form an object. The object is made up of the components that are assembled together, so it must be larger in size than any of the individual components. A part that gets attached in one step of the task might be a lot smaller than the whole object. For example, one step might ask the user to insert a single screw into a large metal engine. Having the user move their head close to the part they just attached, in order to give the camera a better view of how the part was attached, would make it easier for computer vision models to determine when a step has been computed. However, this would be cumbersome for a user.

Instead, the application should work while a user keeps their head in a position that feels natural for completing a task. This requires the system to determine when a step has been completed based on an image that contains most or all of the full object being assembled. We accomplish this using a two stage process, where the system 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.

This process is similar to the two stage process used by Gebru et al. [42]. The first stage involves finding the region of the image that contains the subassembly that a user is working on, using Faster R-CNN [75]. Next, the image is cropped around this region, and the cropped region is classified using Fast MPN-COV [61]. There is one Fast MPN-COV model per subassembly. The Fast MPN-COV model has one label for each step of the task that is part of this model's subassembly. The classification result therefore indicates the step of the task that is shown in an image. The application considers a step to be complete when an image from the camera feed is classified as the label corresponding to the next step. The architecture for our applications is shown in Figure 3.3. Chapter 6 compares the accuracy of two stage process with single object detection and image classification DNNs.

### 3.2.1 Re-Using Labels

A single label may correspond to multiple steps of a task. For example, a kit might contain two identical subassemblies that get assembled on their own, before being connected to the rest of the kit. The steps required to assemble both of these subassemblies will be identical. The subassemblies do not get connected to the rest of the kit until after they have been assembled, so there will not be any visible differences while the user is assembling one or the other. The application can therefore use the same sequence of outputs from a computer vision model for both subassemblies. However, two consecutive steps cannot share the same label.

The application considers a step to be completed when images are classified with the label corresponding to the next step. If the next step had the same label, the application would think that the user completed a step immediately after the step was started. For example, imagine that a developer trained a fine-grained image classifier that had a label corresponding to two curved metal bars attached together. Creating an application with two consecutive steps that asked the user to show the attached two bars would not work. The application would see that the second “attached two bars” step was completed immediately after the first one, because the completion of both steps corresponds to the same output from the image classifier. This issue is illustrated in Figure 3.4.

The issue described in this section results from the fact that some visual change needs to occur in order for the application to determine that a step has been completed. A developer with two consecutive identical steps can likely combine the instructions from both steps together into one single step. Another option is to break up the consecutive steps by adding a step in between them that asks the user to clear everything off the table and then show the empty table to the camera. The only requirement is that the completion of each step must cause some visual change to occur, because this is the method the application uses to determine when a step has been completed.

### 3.2.2 Training

We performed transfer learning from pre-trained models, rather than training models from scratch. Our Fast MPN-COV models were pre-trained on ImageNet 2012 [76] and our Faster R-CNN models were pre-trained on COCO 2017 [62]. We used a PyTorch [71] implementation of Fast MPN-COV and a TensorFlow [20] implementation of Faster R-CNN.

### 3.2.3 Error Correction

Developers can train fine-grained classifiers to recognize specific mistakes that a user of a WCA application might make when trying to complete a task. An error state requires training data, the same way other steps of the task do. When a frame gets classified as depicting an error state, the user is given instructions about how to correct this. Chapter 5 provides further discussion about handling errors with WCA applications.

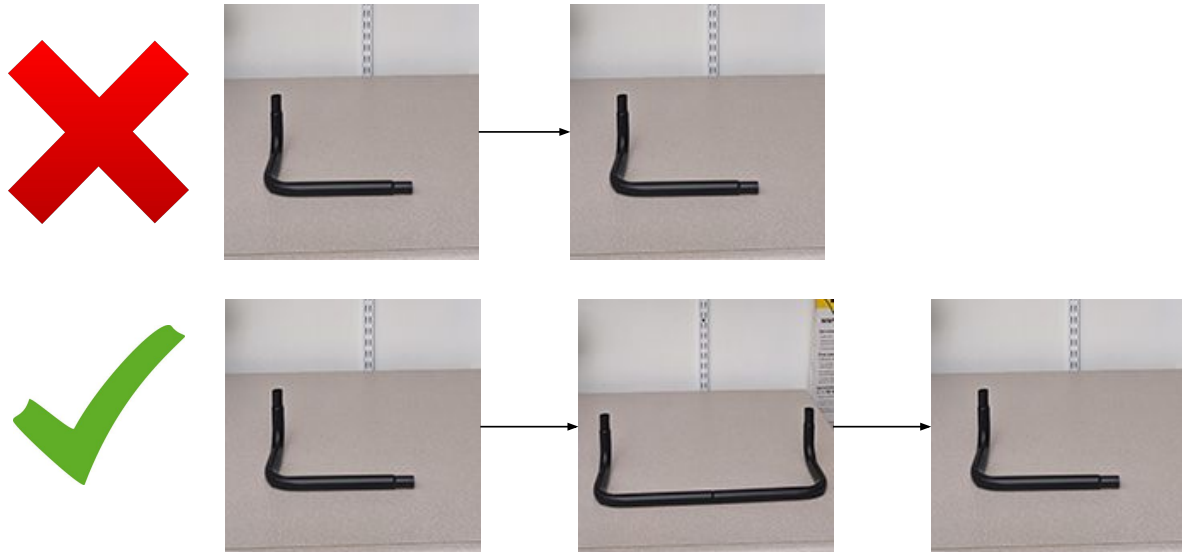


Figure 3.4: The issue with identical consecutive steps. The sequence of steps depicted in the top row cannot be supported by our techniques because the two consecutive steps are identical. There is no visually apparent difference between the two images in the top row. The sequence of steps depicted in the bottom row is acceptable because the identical steps are not consecutive. The image in the middle is different from the images on the right and left.

### 3.2.4 Development Tools

We expanded the Ajalon tools [72] to support the two stage process described in §3.2. Ajalon previously only supported a single object detector, which was sufficient for toy examples such as the sandwich described in [30]. However, more complex assembly tasks require the use of multiple object detectors and multiple fine-grained image classifiers. Our improvements to Ajalon allow developers to have the application use different computer vision models as a user progresses through a task. This results in a single application that will automatically start giving users instructions for the next sub-assembly after they have completed the previous one.

## 3.3 Our Applications

To validate our approach, we developed WCA applications for three real assembly tasks. We trained models for these applications using real videos that were recorded using a smartphone. The videos were manually annotated with bounding boxes using Intel’s Computer Vision Annotation Tool (CVAT). We cleaned up our dataset by computing the perceptual hash values of every image. For all sets of images with identical perceptual hash values, we removed all but one of the images. This resulted in a set of images that all had unique perceptual hash values. We have integrated CVAT and code to remove frames with identical perceptual hash vales into the Ajalon toolchain.



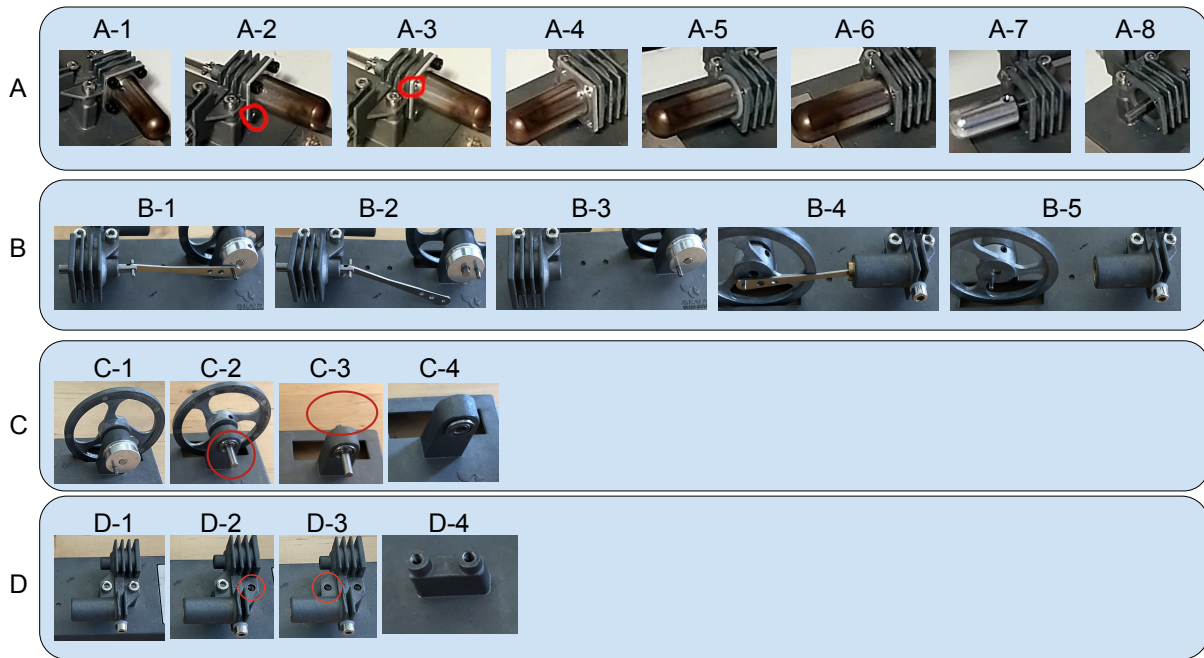


Figure 3.5: The steps detected by our Stirling Engine WCA application. The blue rectangles represent subassemblies. Each subassembly corresponds to a different Fast MPN-COV model.

We have posted<sup>1</sup> all of the artifacts required to run these applications, along with videos showing them being used. We describe each application below in Sections 3.3.1 to 3.3.3.

### 3.3.1 Stirling Engine

The Stirling Engine WCA application guides users through disassembling a Stirling engine. Figure 3.1 depicts the fully-assembled Stirling Engine. 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 these steps have been completed. We split the task into four subassemblies, which are shown in Figure 3.5.

Several steps of the task involved removing screws from the engine. The labels for these steps indicated the number of screws visible in the frame, rather than being unique to the specific step of the task. For example, in Figure 3.6, the first and third steps were both given the label “2 Black Screws.” The training script for Fast MPN-COV randomly flips images horizontally, so we did not want the label to depend on 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. Designing the workflow this way made the computer vision tractable.

We found that illuminating the engine with a table lamp increased the accuracy of the application beyond what we could achieve with overhead room lighting. We lit the object the same way when capturing training data and using the application.

<sup>1</sup><https://cmusatyalab.github.io/roger-thesis/>

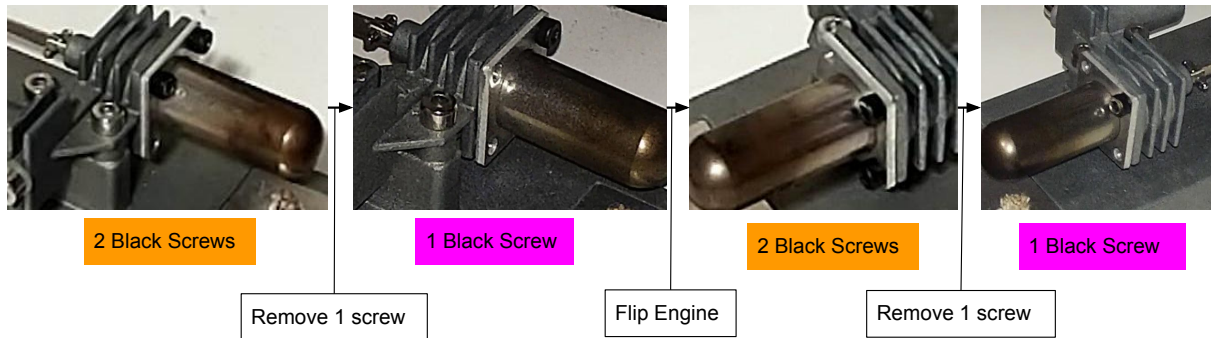


Figure 3.6: Four states from our WCA application for a Stirling engine. The steps look visually similar aside from the number of screws that are visible. The text in colored boxes are the labels that our image classifier was trained on. Note that some different steps were given the same label, but consecutive steps must have different labels. The text in the white boxes describes the actions users take to complete a step.

We created a second version of the Stirling Engine application to guide users through assembling the engine. This version used the exact same computer vision models as our application for disassembly. However, the application requires the labels to be observed in the opposite order, because the order of steps is reversed.

### 3.3.2 Ikea Cart

Our next application provides guidance for users assembling an Ikea Raskog utility cart. The fully assembled cart is depicted in Figure 3.7. The task requires twenty steps to complete successfully. However, the cart has two pairs of identical components that must be assembled the same way. Therefore, four of the steps are repeats of earlier steps. The application uses the same label in cases where steps are identical. Thus there were 16 labels, that each corresponded to the 16 unique steps. In addition, we developed the application to detect one error state, so there were 17 possible labels that our models could output. We split the task into three subassemblies, which are shown in Figure 3.8.

The repeated steps are repeated in pairs. For example, step 1 is performed, followed by step 2. Then both are repeated. Repeating steps in pairs avoids the situation where two consecutive steps correspond to the same label from the classifier.

### 3.3.3 Toy Car

The last application guides users through assembling a model car. The fully assembled model car is shown in Figure 3.9. This task requires 28 steps, which we split into 6 subassemblies. These steps and subassemblies are shown in Figure 3.10. The computer vision models output a unique label for each step of the task.



Figure 3.7: The fully assembled utility cart

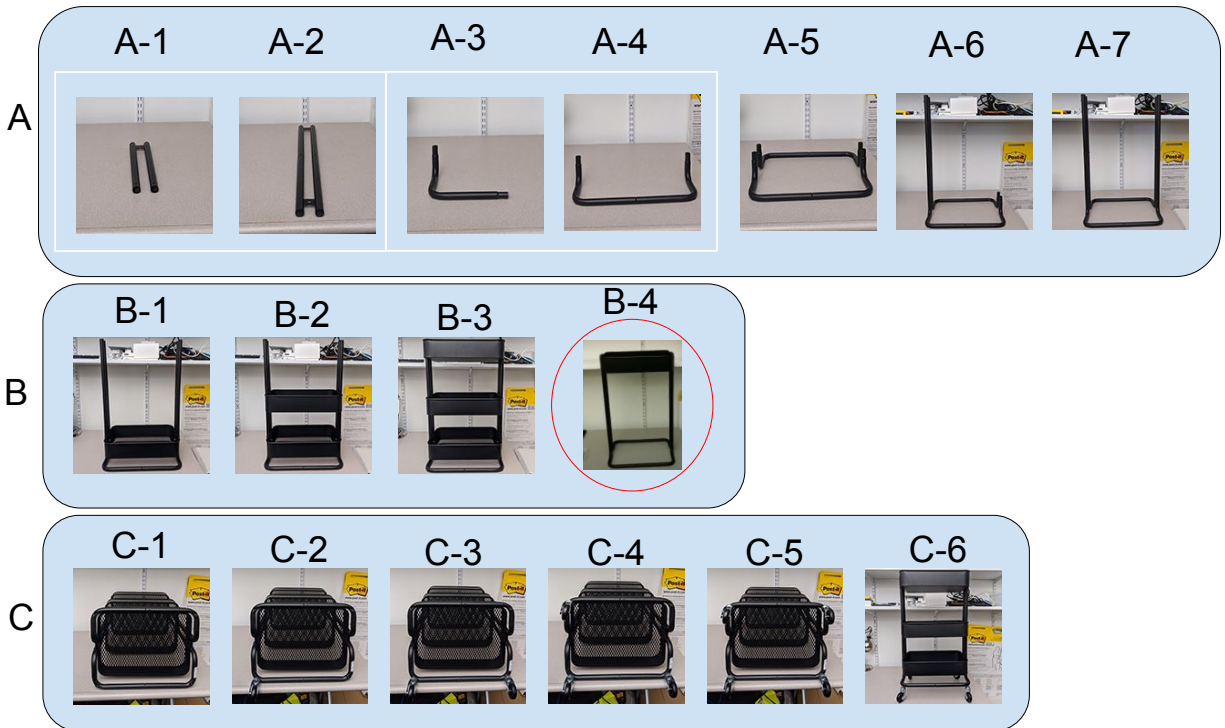


Figure 3.8: The steps detected by our Ikea WCA application. The blue rectangles with rounded corners represent subassemblies. The sequences of steps in white rectangles ([A-1, A-2] and [A-3, A-4]) are repeated. The error state (B-4) appears in the red circle.



Figure 3.9: The fully assembled model car



Figure 3.10: The steps detected by our Toy Car WCA application. The blue rectangles represent subassemblies.

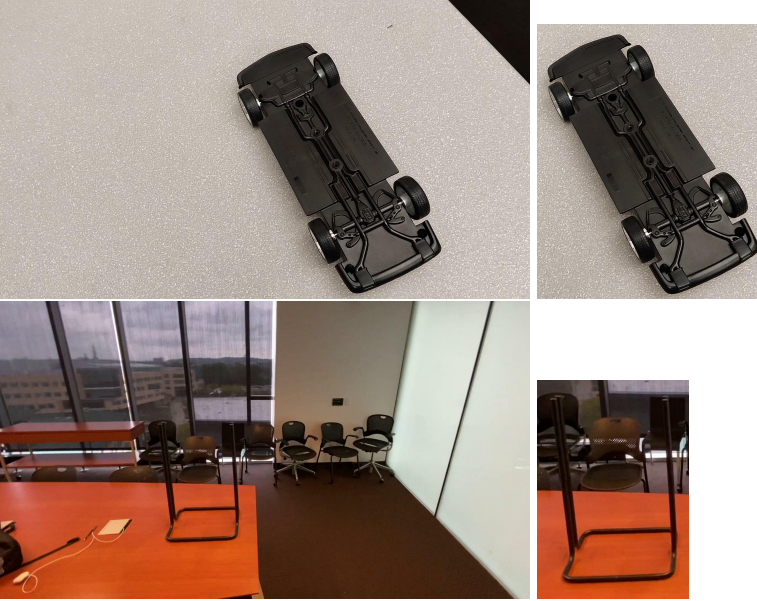


Figure 3.11: Images of the toy car and Ikea cart kits, before and after being cropped

### 3.4 Implementation Details

We captured images at 1920x1080 pixels, and transmitted these to the cloudlet at their full resolution. This is the highest resolution that Android CameraX's ImageAnalysis use case supports. After processing the images with Faster R-CNN, the application crops them around the region that likely contains the object. The cropped image is resized to 448x448 pixels and then classified by Fast MPN-COV. By starting with a large initial image, we ensured that the cropped image would be at least 448x448 pixels. Figure 3.11 shows examples of images before and after being cropped.

The code for many computer vision models is 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 in an interactive application.

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. In addition, it might lead to an application mistakenly believing that a step has been completed before it actually has been. We did not train our models on any



(a) Two Images with identical perceptual hash values



(b) An image with a perceptual hash value that is different from the images in Figure 3.12a

Figure 3.12: Images with identical and different perceptual hash values

images of the assemblies in partially completed states. We experimented with a filter that will only run images through the DNNs when a certain number of consecutive frames have identical perceptual hash values. This essentially means that a certain number of images in a row all had to look very similar. This will only occur when there is no motion in the frame, and the camera is not moving. Figure 3.12 shows examples of images with different and identical perceptual hash values. Requiring a sequence of images to look similar reduced the number of frames that had to be processed using DNNs, and it helped avoid cases where the application erroneously detected completed steps. This technique worked well for WCA applications running on a smartphone mounted to a tripod. But it was less effective for WCA applications running on a Google Glass, due to the motion of the user’s head. Instead, we required a sequence of sequential images to be assigned the same label by the classifier. This helped avoid cases where the application mistakenly believed a step had been completed while a user was still in the middle of it. But it did not save computing resources on the cloudlet, because every frame had to be processed.

## 3.5 Guidelines for WCA Developers

We spent a significant amount of time planning out our applications before we began any development work. The first part of planning involved establishing what each step of the task should be. In particular, a developer should take pictures of what each completed step should look like and write the text for each instruction.

### 3.5.1 Subassemblies

Steps that begin with showing a new part to the camera should be the start of a new subassembly. For example, steps A-1, B-1, C-1, and D-1 in Figure 3.10 all start with a new part. Steps that involve rotating or repositioning the object being assembled were sometimes chosen as the start



of a new subassembly. Steps B-1 and B-4 in Figure 3.5 both involve rotating or repositioning, but we chose to make step B-1 the start of a new subassembly while we did not make step B-4 the start of a new subassembly. None of our applications used a step that involved adding a part to the object being assembled as the start of a new subassembly.

### **3.5.2 Training Data**

It is important to collect training data for each step of the task using a variety of camera angles, backgrounds, and lighting conditions. After training a model, test it on new data that was not included in the training or validation sets. If the performance of the model is not acceptable, create a new training set that includes test images that were classified incorrectly. Then train a new model using this set. The new model should then be evaluated on a different fresh test set that has not been used for anything.



# Chapter 4

## Accelerate WCA development with synthetic training images

Training the DNNs that are used by WCA applications for physical assembly tasks requires thousands of labeled images. Capturing and labeling these images requires substantial effort. Bounding boxes must be drawn around the region of each image that contains the object being assembled. The bounding box must then be labeled with the step of the assembly process that is depicted in the image. Collecting and labeling a training set of images is a major barrier to entry for anyone who wants to develop a WCA application for a new task. For example the Ikea Cart application described in Section 3.3.2 had 17 different states. Labeling images to train the models used by this application took over 50 person hours using Intel’s Computer Vision Annotation Tool (CVAT). This time-consuming and labor-intensive aspect of WCA is the biggest bottleneck to its widespread adoption. Enabling developers to build WCA applications without collecting or labeling training images would significantly help make WCA applications practical for real world tasks.

Previous papers have proposed the use of synthetically generated images for training sets. In this approach, pre-labeling is done by construction [51, 85, 47, 50, 73, 84, 34]. Since programs that generate synthetic images have information about the objects that are visible and their locations, there is no need for manual input of this information. In addition, synthetic images of objects can easily be rendered in a wide variety of different lighting conditions and environments. In contrast, capturing real images of objects in a variety of conditions requires the images of the object to be captured in every such environment. Overall, the use of synthetic data may save considerable manual effort.

This chapter presents our experience training DNNs for WCA applications for three assembly tasks, using synthetic training images.

### 4.1 Meccano Erector Kit

In this section, we ask how well synthetic images work for creating a WCA application for assembling a Meccano Erector Kit. We use the following procedure to answer this question. We first generate a set of synthetic (pre-labeled) images using the Unity Perception package [27]. Unity is a video game engine that includes 3D graphics rendering capabilities and CAD tools. The Perception package was created by Unity to enable the creation of object detectors from



Figure 4.1: Examples of synthetically generated images

CAD models instead of real images that had to be labeled with bounding boxes. This package can create thousands of images, and it will render the subassembly in a different location in each one. It outputs a label file with each image, that contains the coordinates that the subassembly in that image was rendered at. This eliminates the need to manually label images with bounding boxes.

After generating the synthetic images with Unity, we train computer vision models on this data. Next, we collect and manually label a set of real images for the same task, and then train computer vision models on this data. Finally, we compare the accuracy of these two families of models on a held-out test set of real images. Our results show that models created with a training set size of 75,000 synthetic images perform slightly better than models created with roughly 15,000 real images. However, this ordering is reversed when fewer synthetic images are used for training.

The Meccano Kit assembles into a model bike. The fully assembled model is depicted in Figure 4.2. It is made from over 50 parts. However, to limit the amount of work required to separate the CAD model into subassemblies, this work will only separate the bike into three subassemblies. The fine-grained classifiers we trained for this kit had 5 output labels. Three of the labels were for the individual subassemblies, and the remaining two were for the steps required to put the subassemblies together.

### 4.1.1 Generating Data

We found a computer-aided design (CAD) model for the Meccano kit on the community website GrabCAD<sup>1</sup>. This CAD model appears to have been created to replicate the physical Meccano pieces, rather than being the same model that was used to manufacture the pieces. In particular, we noticed a number of differences between the CAD model and the actual Meccano parts. We selected textures for each part of the model, trying to match the appearance of the physical object as closely as possible. We generated synthetic images using the Unity Perception Package [27]. The default setup for this package fills the background of the images that are generated with objects that the network should learn to ignore. Figure 4.3 shows an image generated using this default setup.

<sup>1</sup><https://grabcad.com/library/meccano-9550-002-1>



Figure 4.2: The fully assembled model bike



Figure 4.3: A synthetic image showing part of the bike model. The background is filled with distractor objects that the network should learn not to identify. The background objects are part of the Unity Perception package, and were not customized for the Meccano kit.



Figure 4.4: Bounding boxes with and without padding. The white bounding box has no padding. The Unity perception package uses bounding boxes without padding. The green bounding box has padding. Our dataset uses bounding boxes with padding.

The Unity Perception Package allowed us to make some of the individual parts of the CAD model invisible. This enabled us to generate synthetic images for each step of the task. For a given step, we specified the parts of the subassembly that were visible. We were then able to generate thousands of synthetic images for this step. The perception package generated a label file for each synthetic image, that specified the coordinates of a bounding box around the subassembly and a label name corresponding to the step of the task that was shown in the image.

We trained a Faster R-CNN object detector using this data. The Unity package creates a file with bounding box and label information, and we converted this to the format used by the TensorFlow Object Detection API. The perception package drew bounding boxes tightly around the objects. We added padding to these bounding boxes, to make them more like our hand-drawn labels (which also had some padding). Figure 4.4 shows bounding boxes with and without padding. Training the object detector on images with padding resulted in the object detector returning bounding boxes with some padding. This resulted in higher intersection over union scores when evaluating our object detectors on test data with hand-drawn labels.



Figure 4.5: Our first attempt at making our synthetic images look more realistic. This image is meant to look like an object sitting on a wood floor. The models need to work on real images, so the synthetic images should look as realistic as possible.

Unfortunately the training process for this model did not converge. This might have occurred because it is difficult to differentiate the object we want to detect from the brightly colored distractor objects in the background. We attempted to fix this issue by removing the background objects from the image, and then we tried to make the objects look like they were sitting on a wooden floor. We accomplished this by placing the object at the bottom of the 3D scene in Unity and texturing the floor of the scene with an image of wood from Adobe’s collection of stock images. Figure 4.5 shows one of these images. The Faster R-CNN model trained on this data converged; however, it performed poorly. One issue that we noticed was the object detector mistakenly detected lines in the wood floor as being a model bike assembly. Figure 4.6 shows an example of such an erroneous detection.

We were able to correct this issue by using 15 additional background textures and randomizing the lighting in the scene and the position of the camera. We did not conduct an experiment to determine the minimum number of background textures that are required to achieve good performance. We have posted our code<sup>2</sup>. Figure 4.7 shows some examples of this data. Figure 4.8 shows some of the background images that we used.

After training the Faster R-CNN object detector to find the location of a subassembly, we trained a Fast MPN-COV classifier to determine the step of the task that is shown in the region found by the Faster R-CNN model. Henceforth, we will use the term *model pair* to describe a Faster R-CNN object detector and a Fast MPN-COV classifier created using the same training set. The two stage process was required to achieve acceptable accuracy. Section 6.2.1 examines using a single model instead of the two stage process.

<sup>2</sup><https://github.com/exiaohuaz/data-gen>



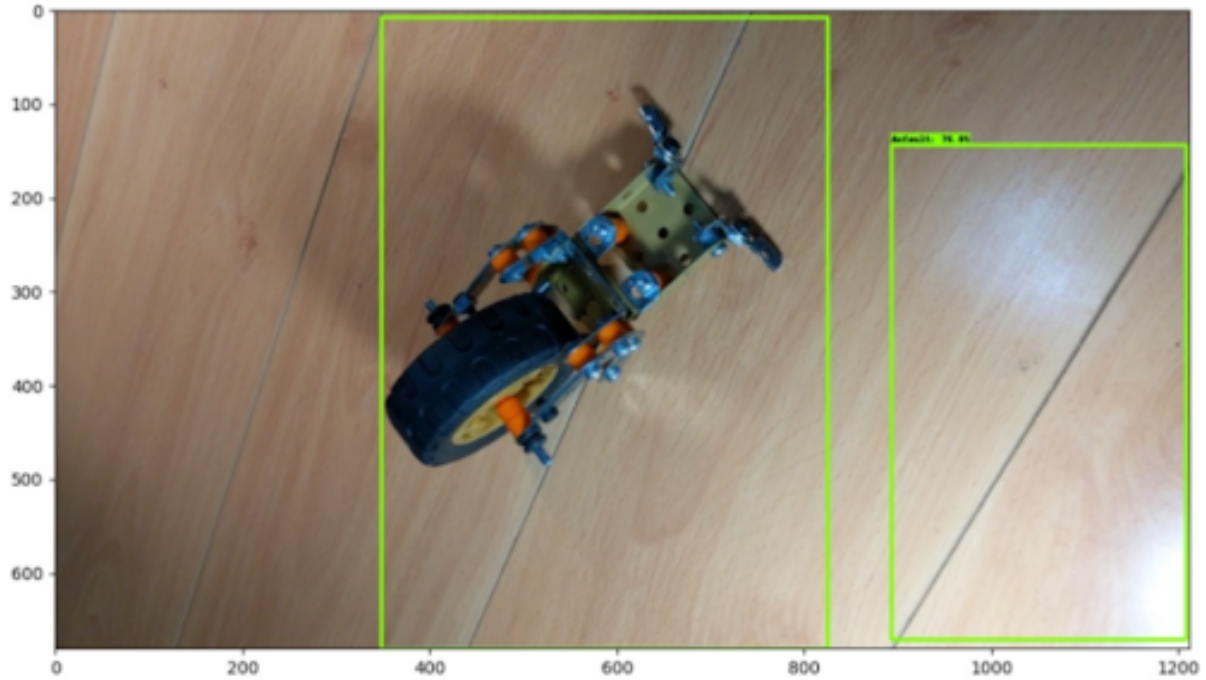


Figure 4.6: Our model incorrectly detected a line in the floor as an object of interest. The green bounding boxes are regions of the image in which the model detected an object.

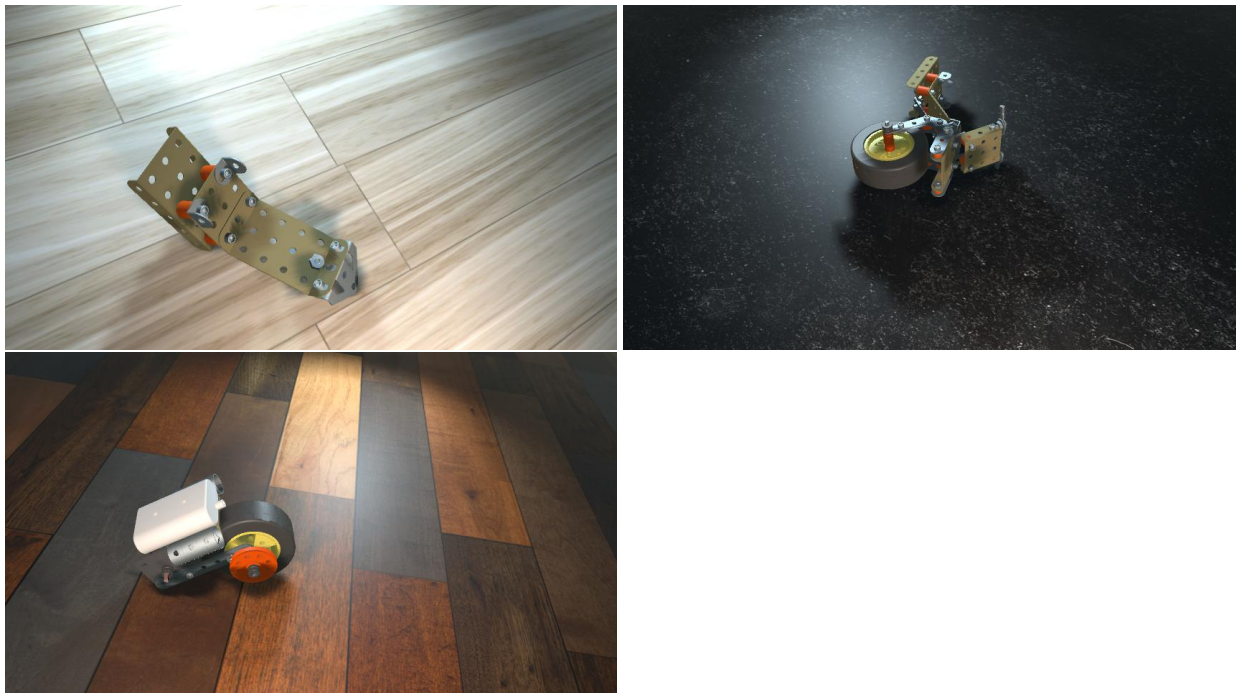


Figure 4.7: Synthetically generated images from our final set. The models trained on this data performed well.

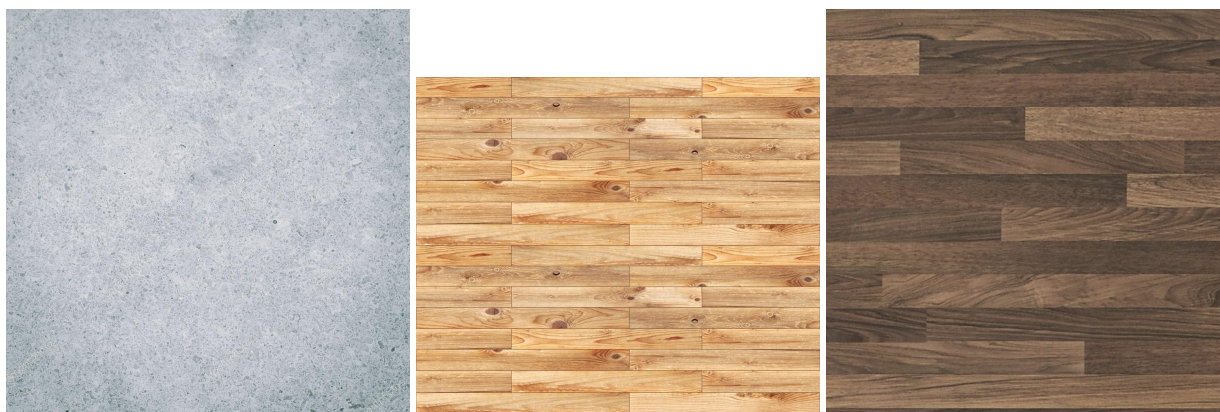


Figure 4.8: Examples Images from Adobe Stock that we used as background textures

## 4.1.2 Results

We evaluated our model pipelines based on accuracy, which is the percentage of images in the test set that were classified correctly. This is equivalent to both top-1 accuracy and micro F1 score. Classification tasks are typically evaluated using top-K accuracy, which looks at the K labels that the model outputs as being most likely. If any of the K labels are correct, the output is considered correct. The value of K is varied based on what seems reasonable for the specific task. Setting K to 5 makes sense for a dataset like Imagenet, with 1000 different labels. However, all of our models were trained on labels for a single subassembly. Each subassembly was built in under 10 steps, so our models had fewer than 10 output labels. In addition, a WCA application can only give a user one instruction at a time. Our classifiers must reliably output the one correct label in order to be useful. We thus chose top-1 accuracy as our evaluation metric, rather than selecting a larger value of K.

All of our training and testing data relates to uncluttered environments with good lighting. We assume that a human using a WCA application can correct environmental issues to reduce classification complexity. For example, the user can increase the amount of light shining on an assembly, or remove clutter from the background. Assuming near-optimal environmental conditions for a WCA assembly task is thus reasonable.

We trained one model pair on real data that was manually labeled with bounding boxes and class labels (15,477 images). The remaining model pairs were trained on synthetic data sets of varying size (12,000, 25,000, 50,000 and 75,000 images). The labels for these images were generated by Unity. We compared the accuracy of these model pairs.

Our test set consists of 4490 real images that are not included in any training set. Table 4.1 presents our results. We observe that the model trained on real data performs better than the models trained on synthetic datasets with 12,000, 25,000, and 50,000 images. However, this relationship is reversed for a model trained on 75,000 synthetic images. Somewhere between 50,000 and 75,000 images lies the cross-over point at which the increased number of synthetic images more than compensates for their lower realism. Changing the quality of the synthetic data, changing the quality of the real data, or changing the amount of real data could change the location of the cross-over point.

Dataset Type	Training Set Size	Accuracy
Synthetic	12,500	69.6%
Synthetic	25,000	79%
Synthetic	50,000	84.1%
Synthetic	75,000	89%
Real	15,477	84.5%

Table 4.1: Classification results for model pairs trained on data for the Meccano kit. Accuracy is the percentage of our 4490 test images that the model pairs classified correctly.

## 4.2 Toy Plane

The next kit we generated synthetic images for was a toy plane that was made up of 3D printed plastic parts. This kit contained six unique parts, and required four steps to assemble. Figure 4.9 shows what the physical kit looks like when it is fully assembled. Figure 4.10 shows the CAD models for the individual parts while Figure 4.11 shows the CAD models for the assembly steps.

We downloaded the CAD file for this kit from the community website Cults<sup>3</sup>, and then 3D printed the kit using this file. We will refer to objects generated from a CAD model that we have access to as being “born digitally.”

We captured real images of the 3D printed parts using a smartphone camera, labeled these images with CVAT [1], and trained a model pair on this data. A review of the labeling confirmed that there were no errors. Next, we generated synthetic training images using the Unity Perception Package. Figure 4.12 contains examples of these images. Afterwards, we trained model pairs on sets of these synthetic images, with varying sizes.

All of our model pairs for the toy plane were tested on a set of 14,996 real images that was separate from any of the images used during training. The results of these tests are shown in Table 4.2. All of our model pairs that were trained on synthetic images performed better than the model pair trained on 39,643 real images. This was different than what we observed with the Meccano kit. The model pair trained on 75,000 synthetic training images of the Meccano kit outperformed the model pair trained on real images of the Meccano kit. However, the model pair trained on real images of the Meccano kit outperformed all of the model pairs trained on fewer than 75,000 synthetic images. One possible reason that synthetic data was more effective for the toy plane than the Meccano kit is that training on synthetic data might be more effective for objects with simpler surfaces. The toy plane is made out of simple plastic, while the Meccano kit is made out of more complex metals. The model trained on 12,500 synthetic images might have gotten high accuracy on our test set because the toy plane’s parts are all made out of non-reflective plastic.

To further investigate why all models trained on synthetic images of the toy plane outperformed the model trained on real images, but the model trained on real images of the Meccano kit outperformed some of the models trained on synthetic images, new training and test sets of real images of both kits should be collected and labeled. The experiments should be repeated

<sup>3</sup><https://cults3d.com/en/3d-model/game/toy-plane-assembled-by-bolts-and-nuts>



Figure 4.9: The fully assembled model plane kit. All parts are 3D printed plastic.

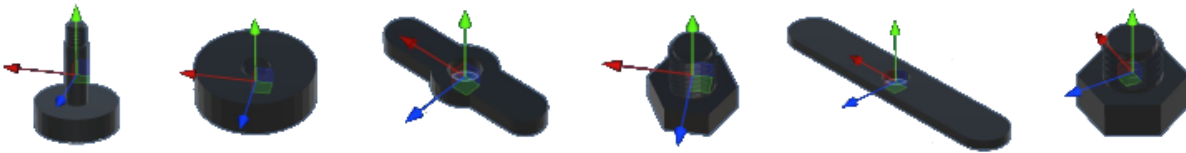


Figure 4.10: CAD models for the individual model plane parts



Figure 4.11: CAD models for the model plane assembly steps



Figure 4.12: Synthetically generated training images for the toy plane kit

Dataset Type	Training Set Size	Accuracy
Synthetic	12,500	87.7%
Synthetic	25,000	89.3%
Synthetic	50,000	90.0%
Real	39,643	76%

Table 4.2: Classification results for model pairs trained on data for the toy plane. Accuracy is the percentage of our 14,996 test images that the pipeline of models classified correctly.

with this new data, in order to see if the same trend still occurs. If the relative performance of the models trained on real data still differs, the experiment should be repeated with a different pair of kits. As with the toy plane and Meccano kit, one kit should be made out of simple plastic, while the other kit should be made out of complex metals.

As we observed with the Meccano kit, the accuracy of our models for the toy plane increased with the size of the training set. However, the increases in accuracy were less dramatic than the increases in accuracy for the Meccano kit, because the model pair trained on the smallest dataset achieved a high accuracy.

### 4.3 Phone Sanitizer

The final kit we generated synthetic training data for was a sanitizer for a smartphone. This kit contained a large metal base and several plastic parts. Figure 4.13 shows the kit fully assembled. The kit contained four unique parts, and there were five steps required to assemble it. Thus there were 9 output labels from the fine-grained classifier.

As with the toy plane, this kit was born digitally, and we had access to the CAD files that the parts were manufactured from. We generated synthetic training images for the sanitizer using the Unity Perception [27] package and Autodesk A3D [89]. Figure 4.14 shows images from the training set that was generated using Unity while Figure 4.15 shows images from the training set that was generated using A3D. Both sets of data were used to train model pairs that were then evaluated on 60,129 real images. As with all data that we labeled in this work, we reviewed



Figure 4.13: The fully assembled Phone Sanitizer



Figure 4.14: Synthetic images of the phone sanitizer, from the Unity Perception Package

all labeling and confirmed that there were no errors. We trained model pairs on different sized datasets, as we did with our other synthetic datasets. The results from these evaluations are presented in Table 4.3. The results from the models trained on images from A3D were noticeably better than the model pairs trained on the Unity data. We again observed that model pairs trained on larger datasets performed better.

	Unity	A3D	A3D with Simple Textures
12,500	75%	89.6%	77.3%
25,000	81.2%	91.6%	76.8%
50,000	85%	92.9%	78.84%

Table 4.3: Classification results for model pairs trained on data for the phone sanitizer. The top row of the table indicates the software used to create the dataset. The left column of the table indicates the size of the training set. The values in the table are the percentage of our 60,129 test images that the pipeline of models classified correctly.

The textures of the sanitizer parts in the A3D images were more realistic than the textures of the parts in the Unity images. In particular, the metal surfaces in the A3D images reflect light more accurately than the metal surfaces in the Unity images. We hypothesized that the realistic textures were responsible for A3D images resulting in models that were more accurate than the Unity images. To verify this hypothesis, we generated a set of images using A3D that used much simpler textures. Figure 4.16 shows examples of these images. We created three different sized

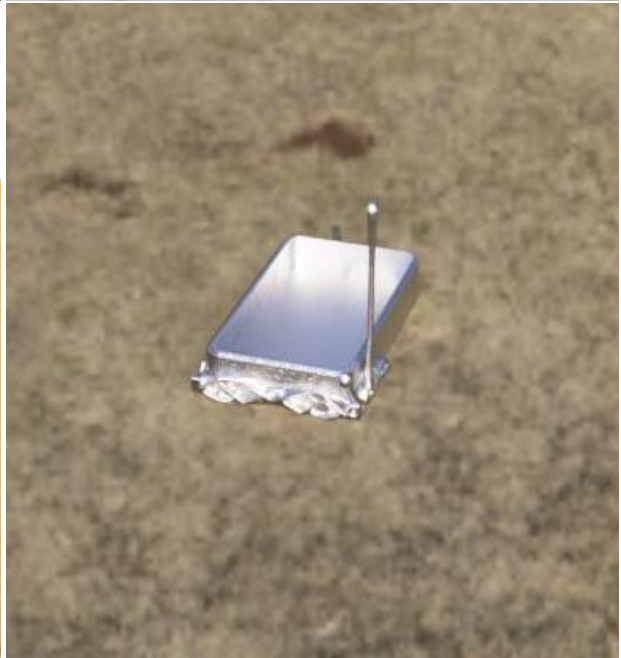


Figure 4.15: Synthetic images of the phone sanitizer, from Autodesk A3D





Figure 4.16: Images generated using A3D with simple object textures

training sets using these images. Table 4.3 lists the accuracies of models trained on these sets, tested on the test set of real images. The accuracy was considerably lower than the models trained on our original A3D dataset, with the same number of images. This supports our hypothesis that realistic textures were responsible for the superior performance of the models trained on the A3D images, when compared with model pairs trained on Unity images.

The model pair trained on 12,500 images was 0.5% more accurate than the model pair trained on 25,000 images. This is a very small difference in performance, but it is unexpected. One possible explanation for this difference is that the 25,000 image dataset contained a set of particularly problematic images that were not in the 12,500 image dataset. The images might have been problematic due to lighting or orientation of the subassembly. These problematic images might have eliminated any benefit that the larger dataset offered. The model pair trained on 50,000 images was more accurate than either of the model pairs that were trained on smaller datasets.

## 4.4 Discussion

Capturing and labeling training images for WCA applications is a time consuming process. Using synthetic images is less labor-intensive, and would simplify development of WCA applications. We have achieved promising results with this approach for three different assembly tasks. These results offer the tantalizing promise that using synthetic data might eliminate the need to manually capture and label images in order to develop WCA applications for assembly tasks.

Wang et al. [89] note several ways that A3D could be improved to make images look better. The first change they propose is increasing the variation in how objects are positioned. Currently, Auto3D renders all objects in a scene with a single texture. Wang et al. [89] propose an improvement that would allow parts to be rendered with different textures. In addition, Wang et al. [89] propose to increase the variation in camera positions used to render the scene. We believe that all of these proposed changes, especially the improvements to camera positioning, could result in better training data for the types of computer vision models that we use. In addition, software tools like Autodesk A3D and the Unity Perception package can be made easier to use. A developer must create individual CAD files for each step of a task. For example, if the models must

recognize a certain kit with a screw inserted and without that screw inserted, the developer must create a CAD file showing the kit before this screw is removed and another CAD file showing the kit after this screw is removed. Open Workflow Editor allows a developer to specify a task using a flowchart. Adding the ability to specify the parts of a CAD model that get inserted or removed during a task step would make it easier to develop WCA applications for kits that are born digitally. In addition, integrating tools for automatic Assembly sequence planning [43] into these programs would improve WCA application development.

After a developer has generated synthetic images, the number of manual steps required to achieve a working WCA application is still very high. Autodesk A3D and the Unity Perception package both output annotations in different formats. The developer must convert these into the format used by the TensorFlow Object Detection API, and create another copy in the format used by the Fast MPN-COV implementation. Afterwards, the developer must start training object detectors and fine-grained images classifiers needed by the application. The developer must then create a flowchart in Open Workflow Editor that specifies the task instructions and the models that should be used at each step of the task. We are actively working to simplify this process by automating the workflow of transformations.

## Chapter 5

# Escalation to Human Experts

The techniques presented in the previous chapters 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 [67].” 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, we handle errors that our models were not trained to recognize by having people completing tasks call a human expert from the application. Detecting all possible error states using the computer vision techniques described in the previous chapters is impractical. However, allowing users to call human task experts to correct errors is practical. Escalation to human experts is thus a necessary component that helps make WCA applications practical for real world tasks.

When a user calls an expert, the user’s camera feed during these calls can be recorded, and these recordings can provide data to train models to recognize these errors in the future. This allows us to follow a DevOps strategy when developing these applications. A team of developers can launch an initial version of a WCA application that detects a small number of error states. As the application gets used, and people call in to get help with new errors, the developers can improve the application to detect these errors. Over time, this process will increase the number of error states that our application can detect.

## 5.1 Experts Without Automation

Several commercial products allow a remote human expert to help a user through an assembly task. Examples of such systems include Microsoft Dynamics 365 Remote Assist [4], Webex Expert on Demand [14], AMA XpertEye [17], and Vieaura [13]. The person completing the task wears a headset with a camera, but the expert must be on a call with a single user for the duration of an entire task. A task expert's time is valuable. We believe that integrating wearable cognitive assistance with human assistance will allow products like these to scale from requiring experts to work one on one through each step of the task to enabling experts to help multiple users concurrently.

## 5.2 Calls With Human Experts

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. The application can also be modified to suggest calling the expert, if a user has been stuck on a step for a certain amount of time. However, additional research is required to determine what should trigger a suggestion to call the expert. Potential options include triggering a suggestion after the application has seen a sequence of frames with identical perceptual hash values, or simply a certain amount of time passing without the user advancing a step. In addition, it might not be necessary to suggest calling the expert in the first place, as users have the option to start the call at any point. Figure 5.1 shows options for task guidance systems that use human experts.

When we train models for WCA applications, we consider each state of a task to be one object. Open World Object Detection [55] 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. WCA 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 WCA 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. This workflow is depicted in Figure 5.2.

We connect users to task experts using Zoom, which offers SDKs for several platforms [19]. The user runs an Android application on a smartphone or Google Glass, which starts a call with the expert using Zoom's Android SDK. The human expert uses a web application that incor-

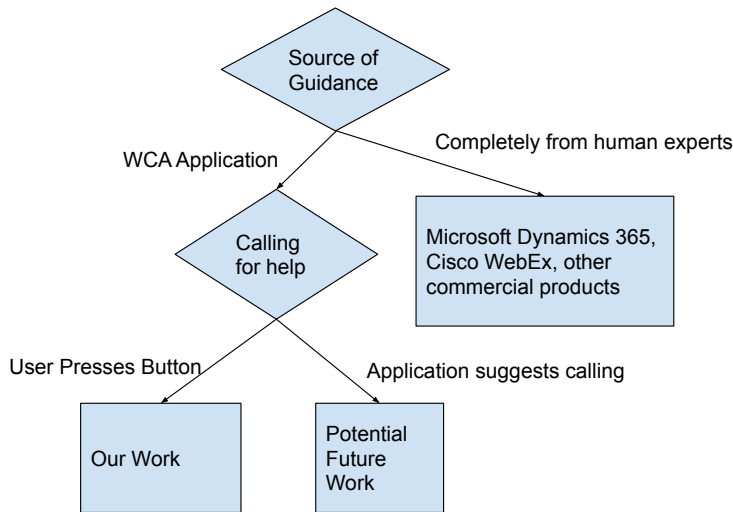


Figure 5.1: The design space of remote expert assistance systems for assembly tasks. Our system primarily guides users with a WCA application. Users must explicitly press a button to start a call with a human expert.

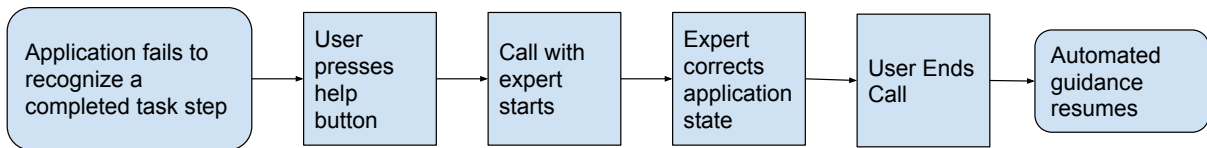


Figure 5.2: The workflow followed to request help from a human expert

porates Zoom’s Web SDK. The components of the system are shown in Figure 5.3. Figure 5.4 shows a screenshot of the application used by the human expert. The expert’s web application allows them to see the user’s camera feed, as well as the step that the user is currently working on. The application works for any WCA task that was created with Open Workflow Editor. The code can be modified to use a different video calling service in the future.

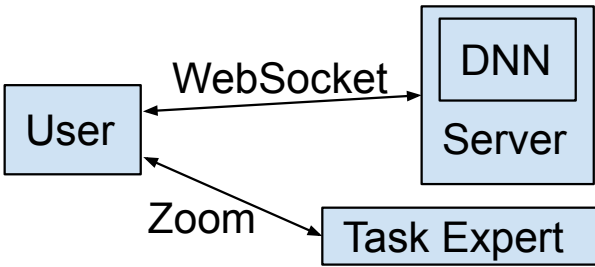


Figure 5.3: The components of a WCA application with human assistance. The user primarily receives guidance from a server running a DNN. If the user reaches a point where the automated assistance fails, they can switch over to receiving guidance from a human expert over a video call.

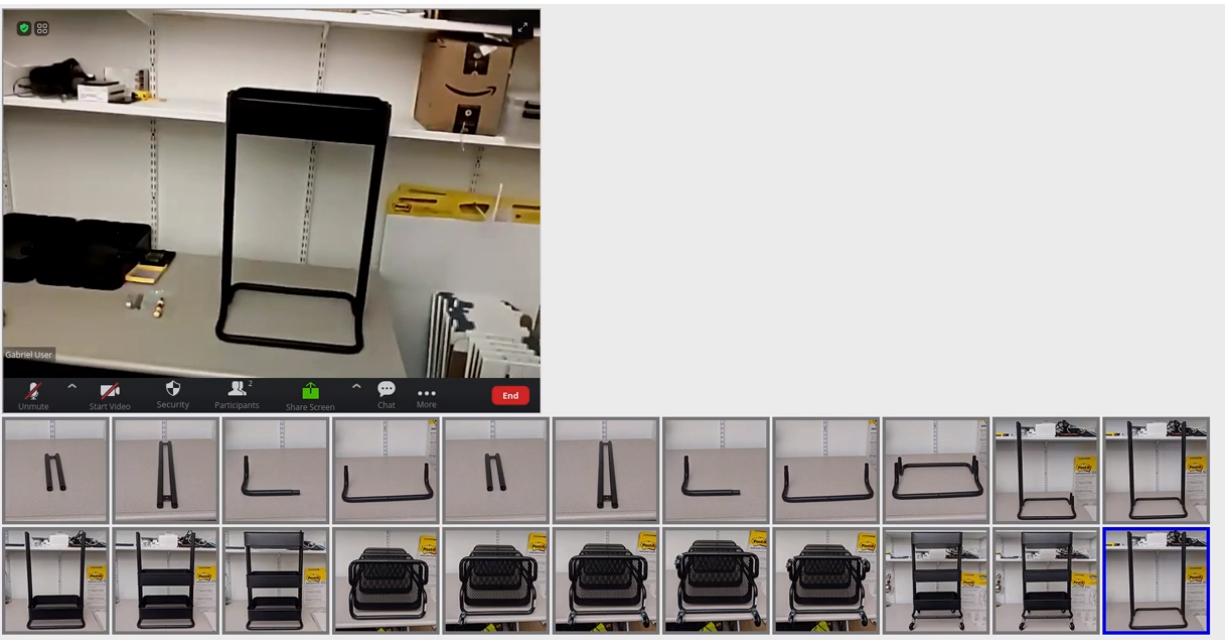


Figure 5.4: A screenshot of the web application used by the human task expert. The feed from the user's camera is shown on top. The task steps are shown on the bottom. The step that the application believes the user is currently working on is surrounded by the blue box. Clicking on a different step will change the current step to the one that was clicked on.

## 5.3 Simulating Call Centers

Supporting a large number of people using WCA applications at the same time would require multiple human experts answering support calls. It is important to employ enough experts to ensure that wait times are reasonable. However, having too many experts working at one time will create unnecessary expenses.

We developed a simulation that people running a call center for a WCA application could use to determine the number of experts they should have available to assist the users who call for help.

There is a large body of work examining wait times for call centers [29, 45]. Our work is different because we simulate a user completing a task with a WCA application. A single user might call the expert multiple times, if they get stuck on multiple steps. In addition, we model user patience, to determine the amount of time a user will wait before calling the expert.

### 5.3.1 A Simple Model

Kendall's notation is used to describe queuing models, by specifying the arrival process, the distribution of service times, and the number of servers [57]. The arrival process determines how the amount of time between calls to the expert should be sampled. The amount of time that a user and an expert spend on a call with each other is the service time. The number of servers refers to the number of experts.

M/M/N is an example of Kendall's notation, where the queue has a Poisson arrival process, a Poisson service time distribution, and more than one server. The time between events for a Poisson arrival process follows an exponential distribution. An exponential distribution has the probability density function  $\lambda * e^{-\lambda x}$  when  $x$  is above 0. As shown in Figure 5.5, it takes the form of a negatively accelerated decreasing function of  $x$ , where the rate of decrease is governed by  $\lambda$ .

The M/M/N model is sometimes called Erlang-C. It has been used to model call centers [29]. Expected wait times for an M/M/N model can be computed using Formula 5.1 and Formula 5.2, which appear in [82]. These formulas compute precise values rather than approximations. Wait time is the period between when a user requests help, and when the user gets connected to the expert. We measure this in seconds. These formulas are a function of the variables listed in Table 5.1.

$$E[\text{Wait for M/M/N}] = \frac{L_q}{\lambda} \quad (5.1)$$

$$L_q = \left[ \frac{1}{(n-1)!} \left( \frac{\lambda}{\mu} \right)^n \frac{\lambda\mu}{(n\mu - \lambda)^2} \right] * \left[ \frac{1}{\sum_{m=0}^{n-1} \frac{1}{m!} \left( \frac{\lambda}{\mu} \right)^m + \frac{1}{n!} \left( \frac{\lambda}{\mu} \right)^n \left( \frac{\mu n}{\mu n - \lambda} \right)} \right] \quad (5.2)$$

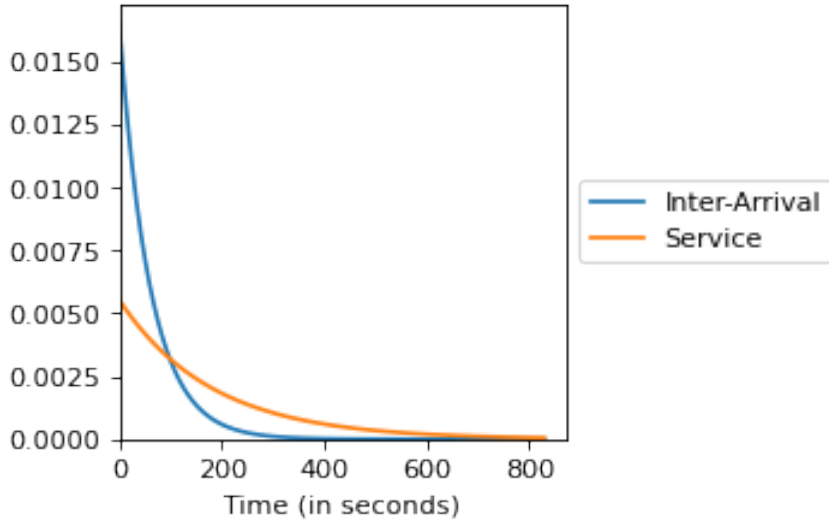


Figure 5.5: The PDFs for the distributions that Simulation 1 samples from. PDF stands for probability density functions. The inter-arrival times and service times were both sampled from exponential distributions.

$L_q$	Expected number of people waiting for an expert
$\lambda$	Average arrival rate. Equivalent to $1/(\text{inter-arrival time})$ . We compute inter-arrival time by taking the average of the inter-arrival samples from our simulation. The units for $\lambda$ are 1/seconds.
$\mu$	Average service rate. Equivalent to $1/(\text{service time})$ . We computer service time by taking the average of service time samples from our simulation. The units for $\mu$ are 1/seconds.
$n$	The number of experts working in the call center.

Table 5.1: The variables used to compute expected wait time for users in an M/M/N queue

### Simulation 1

We compared the wait times from Formula 5.1 with a simple Monte Carlo simulation that we will call Simulation 1. The simulation modeled users calling in, waiting until an expert in the call center is available to speak, and then the user and the expert are on the call for a certain amount of time. There is a single queue for all users waiting for an expert, and it is serviced in first in, first out (FIFO) order. An expert will service the next call from the queue as soon as they finish their current call. The M/M/N queue assumes that all call inter-arrival times are independent of service times and other inter-arrival times. If a user had the option to give up on waiting, this would violate the independence assumption. Our models do not allow the simulated users to give up on waiting. We ran the simulation with different numbers of experts. The experiment was repeated 10 times, with different random values, for each setting of the number of experts.



<b>Inter-Arrival Times</b>	
Distribution Type	Exponential
$\lambda$	1 / 60 seconds
<b>Service Times</b>	
Distribution Type	Exponential
$\lambda$	1 / 180 seconds

Table 5.2: Parameter values for the distributions that Simulation 1 samples from

The inter-arrival time between calls coming in was sampled from an exponential distribution. The lengths of calls were sampled from an exponential distribution with a lower value of  $\lambda$ . These two distributions are depicted in Figure 5.5. Samples were generated using SciPy [86]. Unfortunately we did not have any real data to help inform the parameter values for our distributions. Therefore, we picked parameter values that seemed reasonable based on our experiences with WCA applications. Table 5.2 lists the parameter values used for Simulation 1. Figure 5.6 shows how the waiting times from our simulation and the formula vary as we increase the number of experts. The average arrival rate and average service rate that we plugged into Formula 5.1 were computed based on the inter-arrival times and service times that were sampled by the simulation. The wait times from Formula 5.1 match the wait times from the simulation well. This is expected because Formula 5.1 computes precise values, rather than approximations.

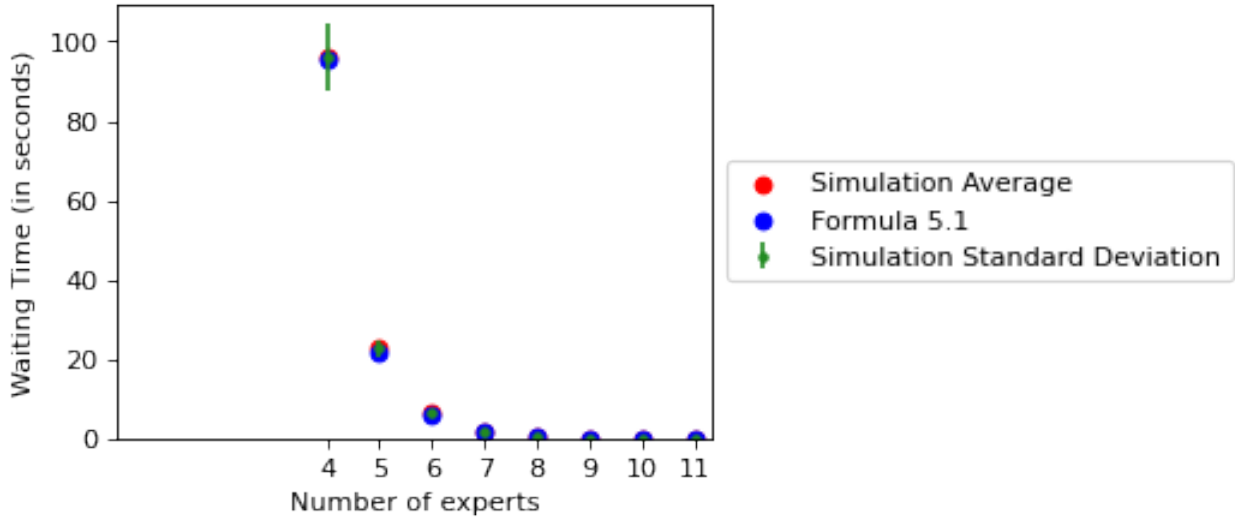


Figure 5.6: The waiting times resulting from Simulation 1 and Formula 5.1. The time values were sampled from the distributions shown in Figure 5.5. The number of users in the system varies as described in Section 5.3.1.

The *system occupancy*, which is computed according to Formula 5.3, cannot exceed 1. Otherwise, calls will arrive faster than they can be answered, and the queue will continue to grow the longer the simulation is run. We thus only report results for cases where the system occupancy is below 1.

$$\rho = \frac{\lambda}{N\mu} \quad (5.3)$$

The number of users in the system at any given time was not fixed. Instead, it is a function of the arrival process, service times, and the amount of time users spend waiting in the queue. Increasing the number of experts while leaving all other parameters the same will decrease time spent in the queue, which will reduce the number of users in the system overall.

### 5.3.2 Lognormal Service Times

Service times are exponentially distributed in the M/M/N model. However, two studies of logs from actual call centers have shown service times to be lognormally distributed [29, 45]. A normal distribution follows a bell curve, with mean  $\mu$  and standard deviation  $\sigma$ . If  $\ln(X)$  follows a normal distribution, this means that  $X$  is lognormally distributed. Figure 5.7 shows an example of a probability density function for a lognormal distribution.

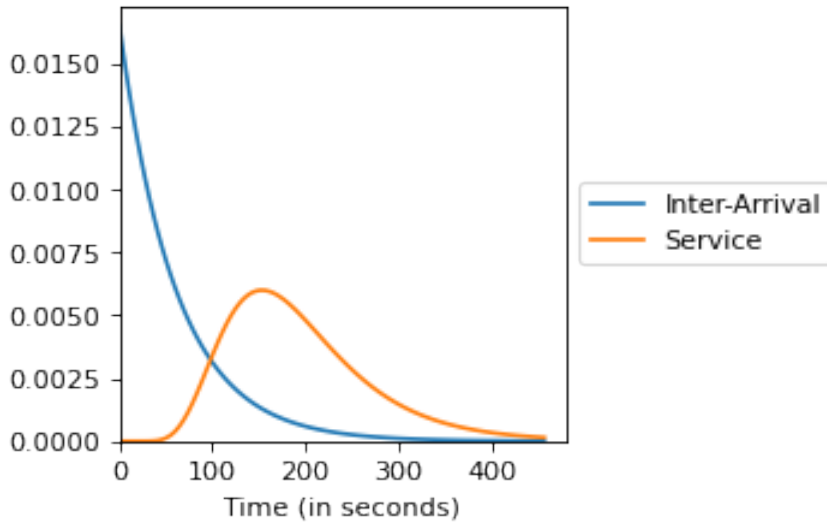


Figure 5.7: The PDFs for the distributions that Simulation 2 samples from. The inter-arrival times were sampled from an exponential distribution. The service times were sampled from a lognormal distribution.

The probability density function for a lognormal distribution is:

$$\frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln(x) - \mu)^2}{2\sigma^2}\right)$$

Lognormal service times require an M/G/N model, which has a Poisson arrival process, more than one server, and allows for any distribution of service times. Brown et al. [29] examined the expected call center wait time using Formula 5.4, which is an approximation of an M/G/N queue.

$$E[\text{Wait for M/G/N}] \approx E[\text{Wait for M/M/N}] * \frac{1 + (\sigma_s * \mu)^2}{2} \quad (5.4)$$

$E[\text{Wait for M/M/N}]$  is computed according to Formula 5.2.  $\sigma_s$  is the standard deviation of service times.  $\mu$  is the average service rate. Note that the average service time is  $1/\mu$ .

<b>Inter-Arrival Times</b>	
Distribution Type	Exponential
$\lambda$	1 / 60 seconds
<b>Service Times</b>	
Distribution Type	Lognormal
$\sigma$	0.4 log(seconds)
$\mu$	log(180 seconds)

Table 5.3: Parameter values for the distributions that Simulation 2 samples from

**Simulation 2**

We modified Simulation 1 to sample service times from a lognormal distribution, but we kept the exponential distribution for inter-arrival time samples. We will refer to this version of the simulation as Simulation 2. The distributions for Simulation 2 are depicted in Figure 5.7, and the parameters of these distributions are listed in Table 5.3. The results from the simulation and Formula 5.4 are shown in Figure 5.8. The average arrival rate and average service rate for Formula 5.4 were computed based on the inter-arrival times and service times that were sampled by the simulation. The waiting times from the simulation were slightly higher than the waiting times from the formula in every case. We believe this difference is due to the fact that Formula 5.4 is just an approximation. In addition, the formula is approximating a queue with an arbitrary probability distribution for service times. Thus it is more general than our simulation, which specifically uses a lognormal distribution for service times.

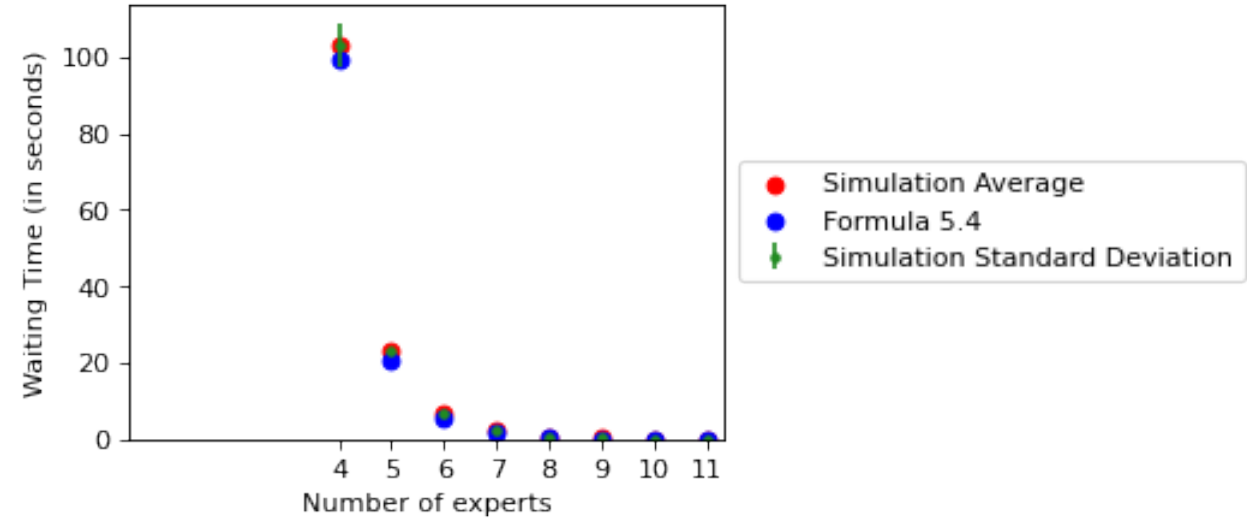


Figure 5.8: The waiting times resulting from Simulation 2 and Formula 5.4. The time values were sampled from the distributions shown in Figure 5.7. The system occupancy was greater than 1 when there were fewer than 4 experts, so we do not have results for these cases for the reasons that we describe in Section 5.3.1. The number of users in the system varies as described in Section 5.3.1.

### 5.3.3 Simulating All Steps

We expanded our simulation to model users completing an entire task with a WCA application. We will refer to this version as Simulation 3. Rather than starting at the point that a call comes into the call center, we also model users receiving automated guidance and calling for help if they get stuck. The simulation includes user patience, which is the amount of time a person is willing to spend on a step, before they give up and call the expert. Most steps of the task will be feasible. This means that a user will eventually complete the step if they spend enough time on it. However, a user's patience is finite. In addition, some steps might be infeasible, which means that the user cannot complete them without calling the expert. Infeasible steps could be a result of incorrectly manufactured parts, bad lighting, or poorly trained machine learning models. It is therefore in the user's best interest to give up on a step and call the expert at some point.

Simulation 3 models users from the point they begin the task. We will refer to the time that the user starts the task as the *entrance time*. The amounts of time in between sequential entrance times (which we will henceforth call inter-entrance times) are sampled from an exponential distribution. The number of users completing the task at any given point is a function of the arrival process, and the lengths of time that it takes users to complete the full task. The simulation models users completing the task according to the process described in Algorithm 1. This process is repeated for each simulated user. The simulation allows users to work in parallel. However, if a user calls for help while all experts are busy, they must wait in a queue for service. As in the previous simulations, all users wait in a single queue that is serviced in FIFO order.

```
for step in task do
  sample patience length;
  sample step feasibility;
  if step feasibility is 1 then
    sample step length;
    if step length  $\leq$  patience length then
      Pause for step length;
      User completes step successfully;
    else
      Pause for patience length;
      User calls expert for help;
    end
  else
    Pause for patience length;
    User calls expert for help;
  end
end
```

**Algorithm 1:** The process simulating one user completing a task using a WCA application

Simulation 3 samples from distributions used in existing literature. Patience length is sampled from a generalized Pareto distribution. Probability theory considers patience lengths to be extreme values [91]. A generalized Pareto distribution can be used to model extreme values.

The generalized Pareto distribution has parameters for location ( $\mu$ ), scale ( $\sigma$ ), and shape ( $\xi$ ).  $\mu$  is equivalent to the mode of a generalized Pareto distribution, rather than the mean.  $\sigma$  is not the standard deviation of a generalized Pareto distribution. The probability density function of a generalized Pareto distribution is:

$$\frac{1}{\sigma} \left( 1 + \left( \xi * \frac{x - \mu}{\sigma} \right) \right)^{-(1/(\xi+1))}$$

Xiong et al. [91] found a generalized Pareto distribution to be a good fit for samples of time that people waited before crossing streets, while the crossing signal was telling them not to cross.

Step feasibility was sampled from a Bernoulli distribution. A Bernoulli distribution models events with two possible outcomes. It uses a single parameter  $p$ , which is the probability of one outcome. Probability values must sum up to one, so the probability of the other outcome is  $1 - p$ .

Step length was sampled from an exponentially modified Gaussian distribution. An exponentially modified Gaussian random variable is the sum of two independent random variables; one that is exponentially distributed and one that is normally distributed. Dawson [32] suggested sampling response times from an exponentially modified Gaussian distribution. Figure 5.9 shows the generalized Pareto and exponentially modified Gaussian distributions that were used.

When the sampled step feasibility value is 1, and the patience length value is smaller than the step length value, the user will call the expert for help. This represents a user giving up on a step that is feasible. One can increase the proportion of steps that a user will give up on by shifting the step length distribution to the right and/or shifting the patience length distribution to the left.

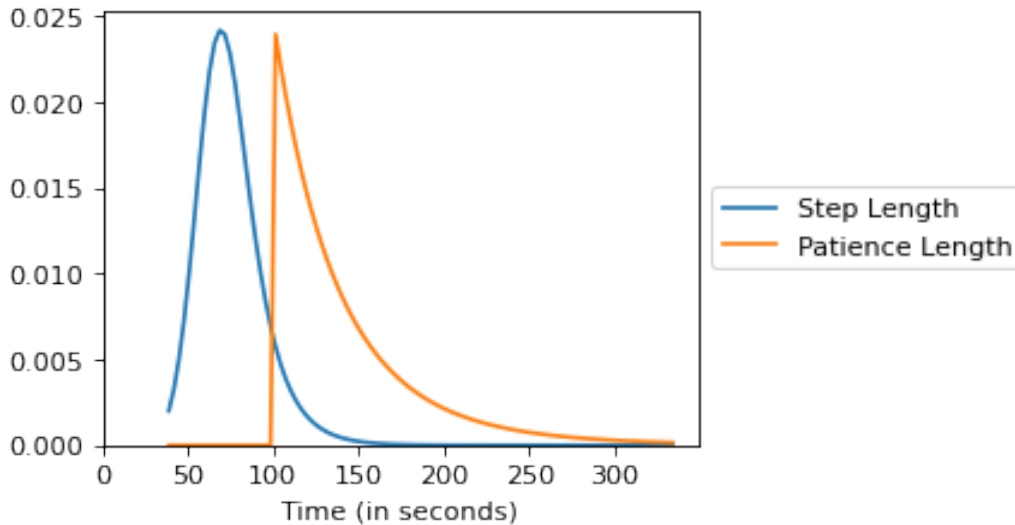


Figure 5.9: The PDFs for the distributions that Simulation 3 samples from. Patience length is sampled from a generalized Pareto distribution and step length is sampled from an exponentially modified Gaussian distribution. Sampling from these distributions resulted in users giving up on about 2.5% of feasible steps.

The samples from the exponential distribution that were used to determine Inter-entrance times for users starting the task are shown in Figure 5.10. The times at which a user in our simulation called for help are shown in Figure 5.11. We fit an exponential curve to both of these sets of data. The  $R^2$  for the exponential fit was 0.99 for the inter-entrance times. This strong exponential fit is expected, because these times were drawn from an exponential distribution. The exponential fit for the inter-arrival times of users calling for help had an  $R^2$  of 0.983. This indicates that Simulation 2, which simply models calls coming in with exponential inter-arrival times, might be accurate enough for modeling call centers for WCA applications.

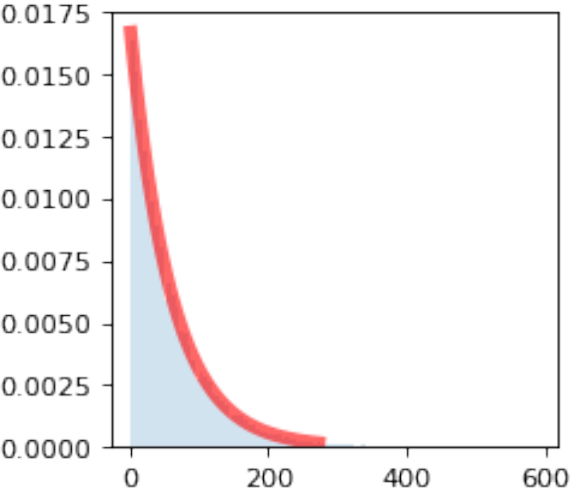


Figure 5.10: Inter-entrance time samples from Simulation 3. The inter-entrance time samples for users starting the task are shown in blue. These were drawn from an exponential distribution. We fit an exponential curve to this data, which is shown in red.

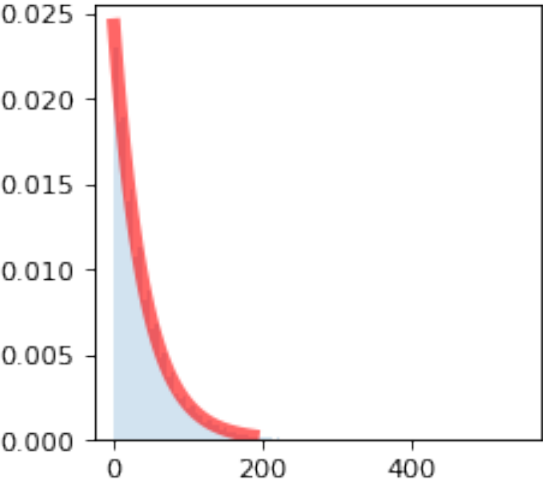


Figure 5.11: Inter-arrival times for help calls that resulted from Simulation 3. We fit an exponential curve to this data, which is shown in red.

<b>Step Lengths</b>	
Distribution Type	Exponentially modified Gaussian
$\mu$	60 seconds
$\sigma$	12 seconds
$\lambda$	1 / 15 seconds
<b>Patience</b>	
Distribution Type	Generalized Pareto
$\mu$	100 seconds
$\sigma$	40 seconds
$\xi$	0.1
<b>Step Success</b>	
Distribution Type	Bernoulli
$p$	0.95
<b>Inter-Arrival Times</b>	
Distribution Type	Exponential
$\lambda$	1 / 60 seconds
<b>Service Times</b>	
Distribution Type	Lognormal
$\sigma$	0.4 log(seconds)
$\mu$	log(180 seconds)

Table 5.4: Parameter values for the distributions that Simulation 3 samples from

## Servicing Calls

Our simulation sampled service times from a lognormal distribution. All of the parameters for the distributions sampled from in Simulation 3 are shown in Table 5.4. The average waiting time for users in our simulation is shown in Figure 5.12, along with the expected waiting times from Formula 5.4. There was a huge variation in waiting time across different runs of the simulation with five experts. However, this variation decreased substantially when we ran the simulation with six experts. This makes sense intuitively, as there was more of a buffer to handle bursts of calls coming in.

The simulation results are plotted against the times from Formula 5.4 in Figure 5.13. Fitting a linear model to these values achieves an  $R^2$  coefficient of 0.86. This fit indicates that the waiting times from the simulation are reasonably well correlated with the waiting times from the formula.

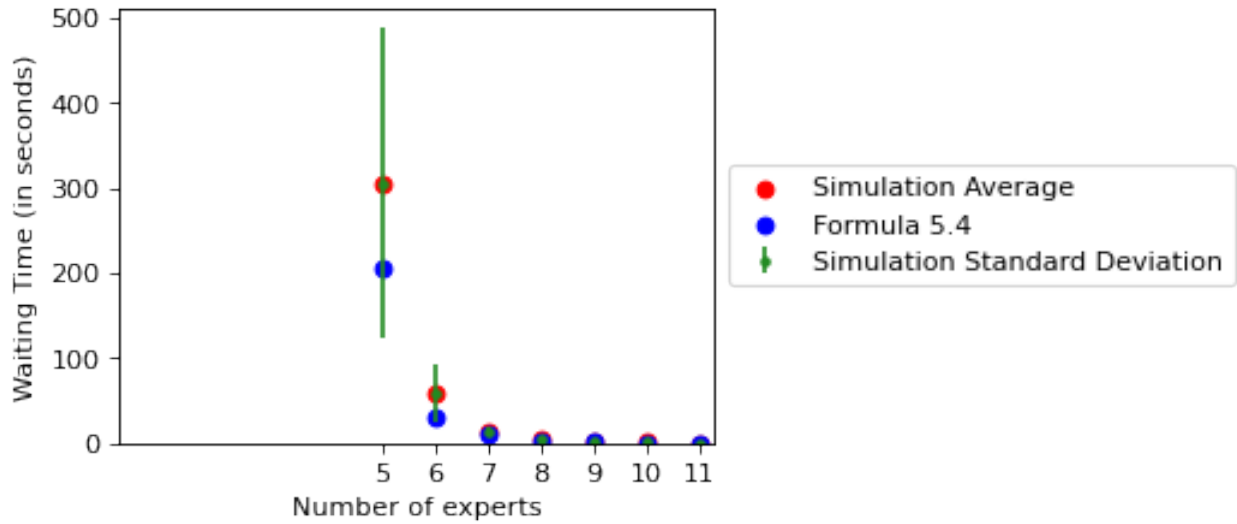


Figure 5.12: Waiting times from Simulation 3 and Formula 5.4. The standard deviation of waiting times from the simulation was large when there were five experts. However, increasing the number of experts decreased this standard deviation. The number of users in the system varies as described in Section 5.3.1.

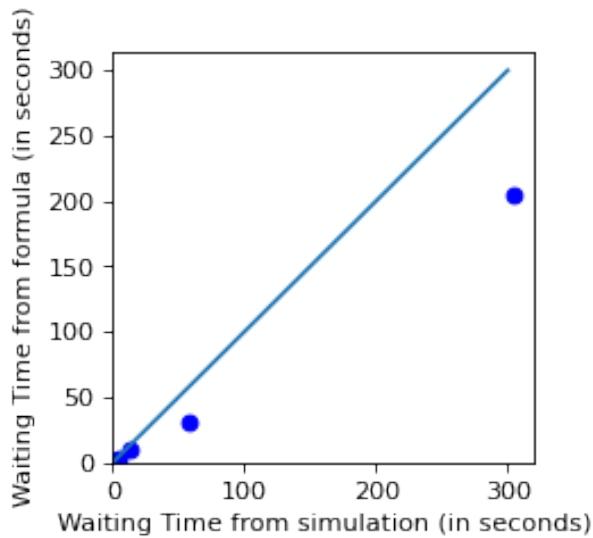


Figure 5.13: Waiting times from Simulation 3 plotted against times from Formula 5.4. The waiting times in Figure 5.12 drop significantly when the number of experts is increased from 5 to 6. Thus there is a large gap between the points in this figure.



## 5.4 Extending to Real Call Centers

We did not have data for real workers completing WCA tasks and calling experts for help. We therefore chose parameter values for the distributions that our simulations sampled from based on our experiences developing and using WCA applications. In order to make our work helpful for a practitioner staffing a call center for a real WCA application, we developed a tool that runs our simulations based on real data for a specific WCA application. In order to use this tool, a user inputs data samples collected from a call center. The required samples are lengths of task steps, lengths of user patience, lengths of calls, inter-entrance times of workers, and the probability of a user completing a step correctly.

The tool is available as a Google Colab notebook. Google Colab is a web based service for running Python code. A Colab notebook can also be downloaded and run locally as a Python script or a Jupyter notebook. We have posted the tool publicly<sup>1</sup> under Version 2.0 of the Apache License. All of the source code is available at that link. There is a version of the tool that allows users to copy and paste data samples into a web form, and a second version that allows users to upload data as CSV files. After a user uploads data, the notebook fits distributions to this data and runs the simulation with samples from the newly-fitted distributions. The notebook generates a graph similar to Figure 5.12.

## 5.5 Exploring Parameter Space

This simulation allows us to see how average wait time changes, when we modify one variable and leave all other variables constant. Although the quantitative outcomes will depend on the specific source distributions, which were parameterized here based on observation of previous WCA applications, the results of these simulations illustrate the pronounced increases in wait time that occur given a shift in critical parameters, such as greater incidence of infeasible steps, reduced user patience, or inclusion of more long steps that lead users to call on the expert.

We ran these simulations with nine experts. We sample from the same distributions used in Simulation 3. The simulation was run ten times for each variable value we tested.

### 5.5.1 Varying the Proportion of Feasible Steps

This experiment varied the proportion of steps that a user could complete. A feasible step is one that a user can complete if they spend enough time on it. However, a user might choose to give up and call the expert before the step is complete. Users are guaranteed to complete a feasible step if they spend enough time on it. However, the user might actually give up and call the expert before they have spent the time necessary to complete the step. An infeasible step cannot be completed unless the user calls the expert. Results are shown in Figure 5.14. As the proportion of infeasible steps increases, the average waiting time increases. This is because an increase in the number of infeasible steps will increase the number of calls to the expert.

<sup>1</sup><https://cmusatyalab.github.io/roger-thesis/>

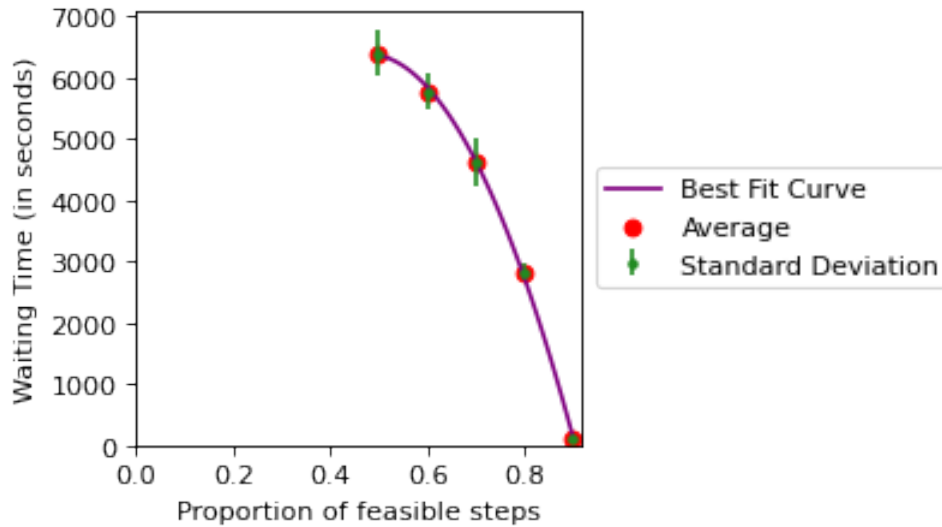


Figure 5.14: Average wait times resulting from changing the proportion of feasible steps. These wait times were fit by a second-order polynomial as shown, with an  $R^2$  value of over 0.99. This indicates a steep cost as a WCA application is penetrated by infeasible steps, for example, due to ineffective image processing or manufacturing error. As we describe in Section 5.5.1, feasible steps are those that are completable, but a user may still give up on a feasible step and call the expert.

### 5.5.2 Varying Patience Length

Patience length is the amount of time that a user is willing to spend trying to complete a step, before calling the expert. The average waiting time that resulted from different average lengths of patience are shown in Figure 5.15. Increasing average patience length will increase the number of steps that users will complete on their own without calling the expert. The number of calls to experts therefore decreases. Thus the average waiting time decreases as average patience length increases.

### 5.5.3 Varying Length of Feasible Steps

This experiment varied the average amount of time that a user must spend in order to complete a feasible step. Figure 5.16 shows the results of this. Increasing average step length increases the likelihood that a step will take longer than a user's patience length. This results in more calls being made to the expert. Therefore, increasing average step length increases the average waiting time.

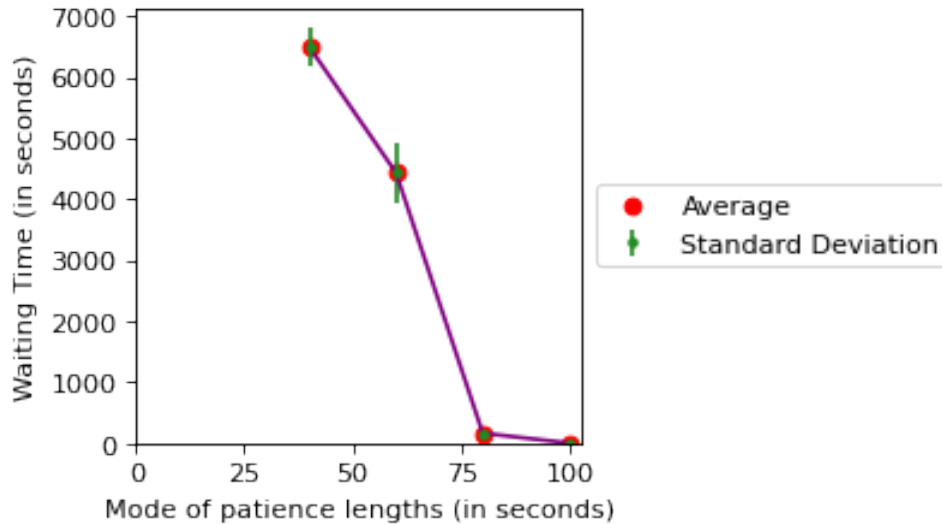


Figure 5.15: Average wait times resulting from changing users' simulated patience. Given the current parameters, with nine experts available, the waiting times move from a scale of minutes to hours when the patience length is doubled from 40 sec to 80 sec. A Generalized Pareto distribution, which is what we use to sample patience lengths, is parameterized by its mode. This is thus the parameter that we vary, in order to change patience length samples.

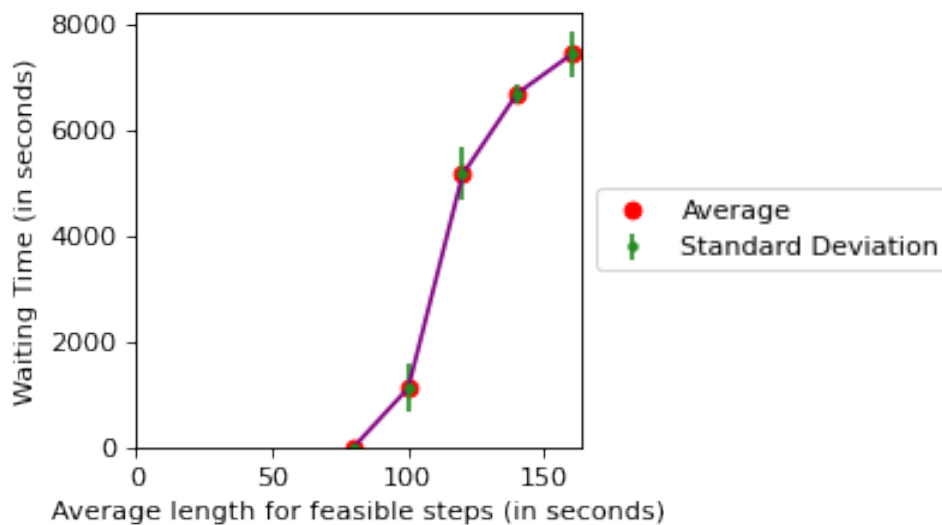


Figure 5.16: Average wait times resulting from changing the average step length. Step length refers to the amount of time it took to complete a step that was feasible. See Section 5.5.1 for how we define a feasible step. Given the current parameters, with nine experts available, doubling the average step length from 80 sec to 160 sec moved waiting times from essentially no waiting to over 2 hours.

## 5.6 Limiting the Number of Active Users

WCA applications might be licensed for use by a certain number of people at any given time. We will henceforth use the term “active users” to refer to the number of people using an application at a given time. For example, an organization might buy a license for a WCA application that supports 10 active users. If 10 people from this organization are using the application, and an 11th person tries to start using it, the 11th person will be given an error message saying that the organization only paid for a license for 10 active users. This error message will prevent the application from starting.

This licensing strategy has the advantage of limiting the resources required to support application users. The licenses could be specific to a particular location. In this case, a provider could provision cloudlet resources at that location in order to support the number of active users that a license allows. In addition, limits on the number of active users make it easier to decide how many human experts to have available in the call center.

We created a version of our simulation that included a limit on the number of active users. We will henceforth call this version Simulation 4. Simulation 4 is identical to Simulation 3 while the number of active users is below the limit. Once the number of active users reaches the limit, new users will be unable to start the task. One of the active users must complete the task, which will cause the number of active users to drop below the limit, before a new user will be allowed to begin the task. Users attempt to begin the task according to the same Poisson arrival process used in Simulation 3. However, if a user attempts to begin the task while the number of active users is equal to the limit, they will leave the simulation without starting the task.

Figure 5.17 shows the average waiting times that resulted from running Simulation 4 with different limits on the number of users. Figure 5.18 shows the number of experts that were required to achieve certain average wait times for users in the queue.

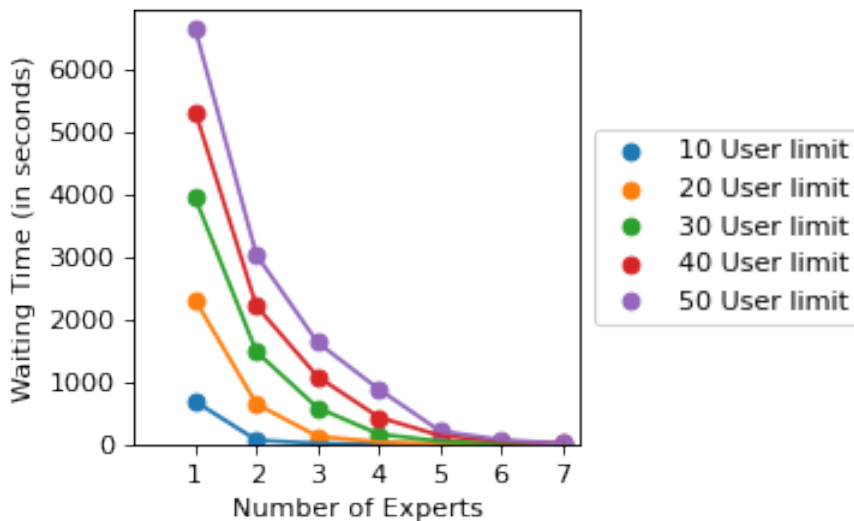


Figure 5.17: Average wait times resulting from Simulation 4. We varied the number of experts and the limit on the number of active users allowed in the simulation.

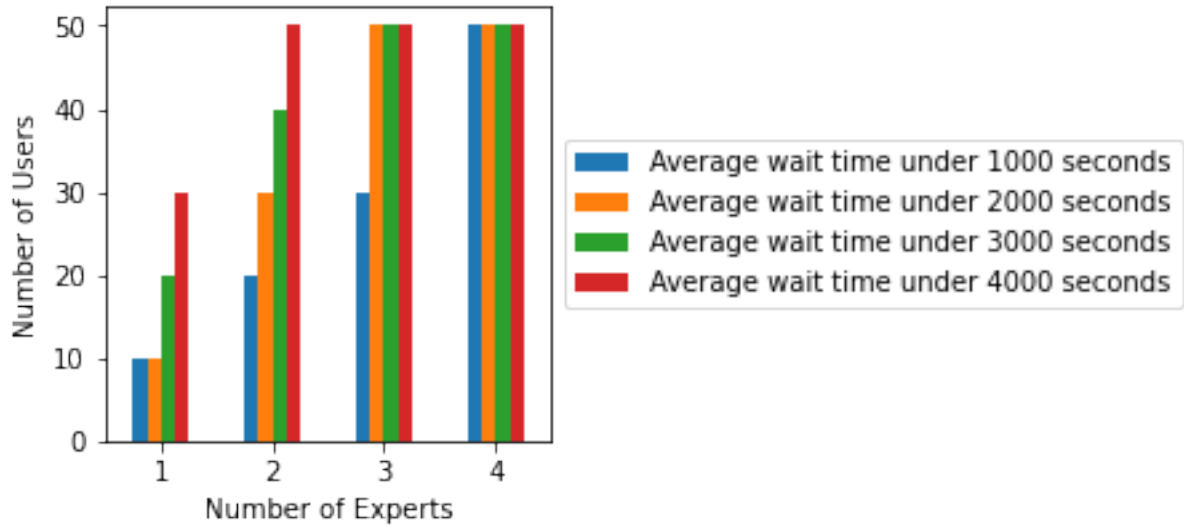


Figure 5.18: The number of users that were serviced under certain wait time thresholds. These numbers were obtained from Simulation 4. As expected, the number of users increased as we increased the number of experts.

## 5.7 Summary

In order to help users correct mistakes that they make completing an assembly task with the help of a WCA application, we added a call functionality to our applications. Users can press a button to request help from a task expert, who can see the user's camera feed and talk back and forth with the user. We developed and evaluated Monte-Carlo based tools to predict the number of experts that must be available to help a given set of users.



# Chapter 6

## Device and Cloudlet Implementation

This chapter describes a new version of the Gabriel software framework that we developed for WCA applications. Next, it examines DNNs that can be run on mobile devices, and how these models can be used in WCA applications to reduce the bandwidth and latency consumed by each WCA application user. The new software framework contains modular components that allow developers to implement WCA applications in a scalable and maintainable way. Reducing bandwidth and latency consumption are critical for the scalability and practicality of WCA applications in real world settings.

### 6.1 Software Framework

We developed a new version of the Gabriel software library for WCA applications [5]. The primary function of this library is to transmit data from mobile devices to cloudlets, and to obtain results in a timely manner for WCA, in spite of networking delays and cloudlet load. 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. [48].

#### 6.1.1 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 in order to support activity recognition in WCA tasks. 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, thereby lowering the implementation complexity of simple WCA applications.

## 6.1.2 Key Abstractions

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 or video, or set of readings from a different type of sensor. Note that a frame refers to a reading from one sensor, not multiple sensors.

All of the WCA 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.

### Flow Control

Our flow control mechanism is based on *tokens*. A token represents a frame that a client is waiting to have processed. A cloudlet operator sets the number of tokens that are given to a client. Clients have a set of tokens for every source. For example, if a client has two sources, and the number of tokens on the cloudlet is set to 1, the client will be given one token for each of the two sources. The cloudlet operator sets the number of tokens based on the amount of latency that is acceptable for a given application. 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 [16], 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 is idle. It also reduces the number of dropped frames. The number of tokens is thus a parameter whose increase will increase the framerate for applications that can tolerate higher latency.

Consider a network with latency so high that transmitting a frame from a client to the cloudlet takes longer than processing the frame on the cloudlet. If the number of clients is low, the cloudlet



operator can increase the number of tokens that clients are given. This will cause a queue of unprocessed frames to form on the cloudlet. It will also cause clients to send a larger number of frames to the cloudlet, rather than dropping them. However, the amount of time between a client sending a frame, and receiving a result back for the frame will increase. It is therefore advisable to run latency sensitive applications, such as WCA, with a single token per client.

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 an RGB camera, these frames are put into the same queue. Figure 6.1 shows an example of how frames are inserted into queues.

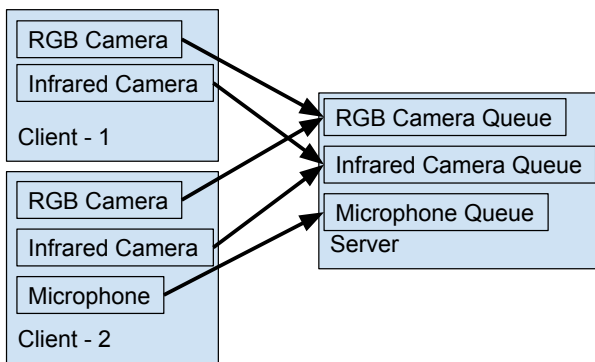


Figure 6.1: Two Gabriel clients that produce frames from multiple sensors. The arrows represent frames being inserted into queues on the server.

One of our design goals for this new version of Gabriel was to ensure that frames get consumed at the rate that the fastest cognitive engine can process them. An instance of Gabriel running with a single cognitive engine that processes camera images at 30 FPS should return tokens to clients at a consistent rate, with or without a second cognitive engine that processes frames at 15 FPS. In other words, the addition of a slower cognitive engine should not change anything from the client's perspective. The rate tokens are returned is the same if Gabriel is run with one fast cognitive engine, or if it is run with the same fast cognitive engine and a second cognitive engine that is slower.

Our second goal was to ensure that cognitive engines do not get stale frames because they are slow. If a fast engine processes multiple frames in the time it takes a slow engine to process one frame, the slow engine should not be given all of the frames that were given to the fast engine. This would create an increasingly large queue of stale frames for the slow engine.

We accomplish these goals by having two possible things happen when a cognitive engine finishes processing a frame.

1. If the frame that was processed is the most recently removed frame from the queue, a new frame is removed from the queue and sent to the cognitive engine for processing.

2. If the processed frame is the not most recently removed frame, no new frame is removed from the queue, and the cognitive engine is sent the *existing* most recently removed frame. This process is illustrated in Figure 6.2.

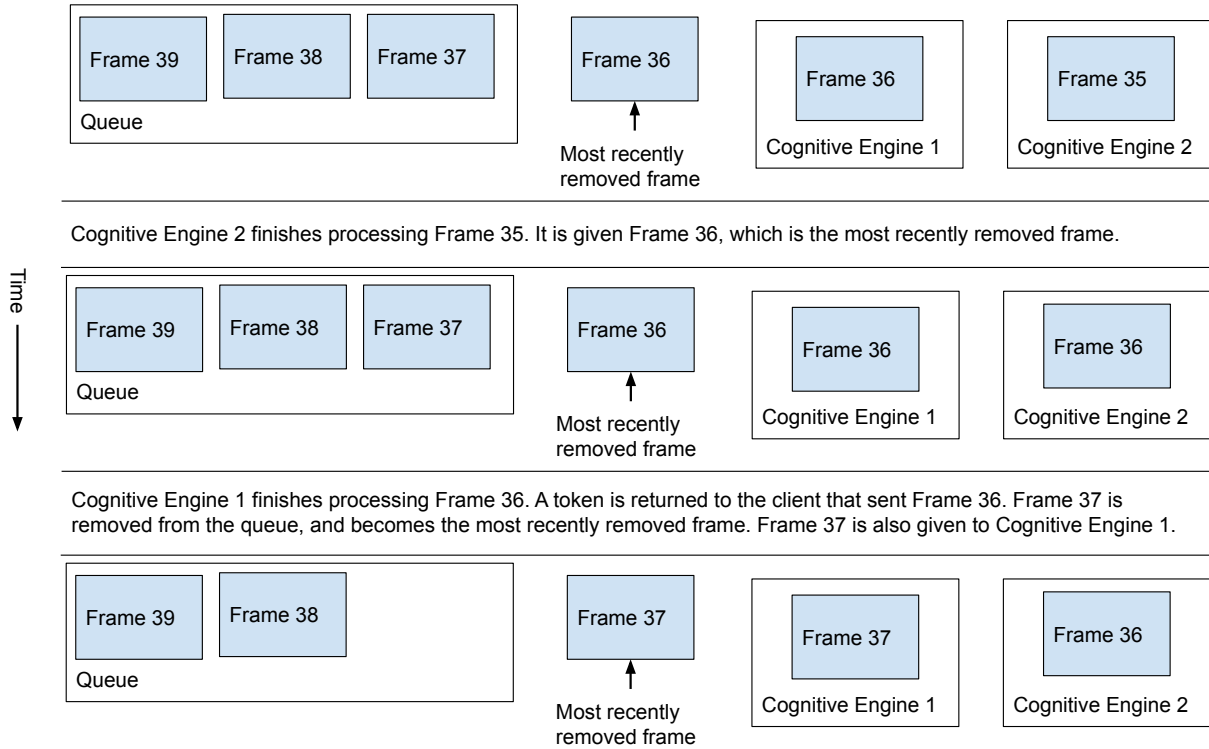


Figure 6.2: Two cognitive engines consuming frames from the same queue

A cognitive engine finishing processing the most recently removed frame represents this cognitive engine being the fastest. However, this process still works if the amount of time that cognitive engines take to process frames varies, and the fastest engine changes. The decision is solely based on whether or not the frame that has just been processed is still the most recently removed frame.

## Workflow

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 6.3), 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 [18]. All Python programs can be run on a single cloudlet, or they can be run on different cloudlets.

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

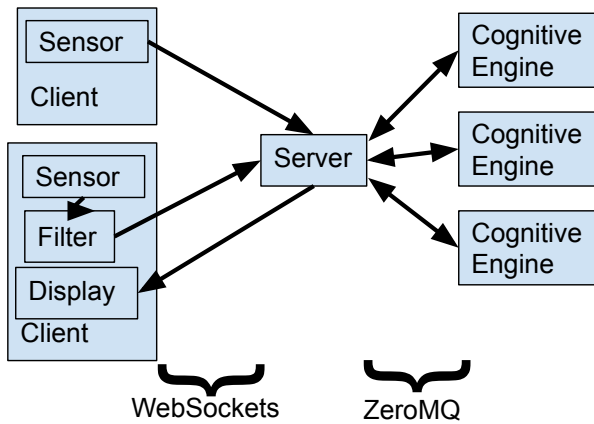


Figure 6.3: A Gabriel workflow with two clients and three cognitive engines

functions to capture images with a camera and transmit the latest frame whenever a token is available. The Python library uses OpenCV [28] to capture images while the Android library uses CameraX [3]. Our Python code has been published to The Python Package Index (PyPI) [10, 8] and our Android code has been published to Maven Central [2].

## 6.2 Leveraging Mobile Device Hardware

This section considers running shifting some (or all) of the computations for WCA applications from cloudlets to mobile devices. This leverages on-device computation to reduce bandwidth and cloudlet usage.

### 6.2.1 Accuracy Comparisons

We compare the accuracy of models and model pairs that developers can use in WCA applications. Some of these models can be run directly on mobile devices or on cloudlets, while others can only be run on cloudlets.

#### Running a Single Model

In an attempt to develop extremely lightweight versions of our applications, we considered using a single DNN, rather than the pipeline described in §3.2. We used data from four of our applications, which is summarized in Table 6.1. The training set contains images that were labeled with a bounding box around the subassembly. The test set contains images that are distinct from the training set, but were not labeled with bounding boxes. WCA applications need to indicate the step of a task that is shown in a camera feed, but they do not need to provide bounding box coordinates for any parts in the image. Our evaluation thus did not examine bounding box coordinates, so we did not label any test images with bounding boxes. All images in the training and test sets were assigned a class label, indicating the step of the task that was shown in the image.

Name	Description	Set Size	
		Training	Test
Stirling	Assemble a heat engine from metal parts	9598	10010
Meccano	Build a model bike from metal parts	15477	4490
Toyplane	Build a model helicopter from 3D printed plastic parts	55000	14996
Sanitizer	Assemble a sanitizer for a smartphone from metal and plastic parts	49956	60129

Table 6.1: A summary of the data used for the experiments in Chapter 6. Set sizes are measured in number of images. Each dataset corresponds to one of our WCA applications.

	Meccano	Stirling	Sanitizer	Toyplane
Resnet 50	69.8%	26.3%	68.3%	56.4%
EfficientDet-Lite0	<b>75.2%</b>	53.7%	79.3%	51.1%
EfficientDet-Lite1	71.1%	53.8%	84.1%	63.5%
EfficientDet-Lite2	<b>75.2%</b>	<b>57.8%</b>	84.9%	59.8%
Fast MPN-COV	73.5%	52.0%	84.0%	<b>78.0%</b>
Faster R-CNN	72.3%	50.7%	<b>91.0%</b>	67.5%

Table 6.2: Classification accuracy for standalone DNN models. Accuracy is the percentage of images that the model classified correctly. The highest accuracy for each application is in bold.

For each application, we trained a Resnet 50 [49] image classifier, three different sized EfficientDet [83] object detectors, a Fast MPN-COV [61] classifier, and a standalone Faster R-CNN [75] object detector. We then evaluated these models on the test set for the relevant application. When evaluating object detection results, we ignored bounding box coordinates and just checked if the class label for the detected object with the highest confidence score was correct. Table 6.2 lists the accuracy of these models.

### Running a Pipeline of Models

The results in Table 6.2 leave a lot of room for improvement. We next evaluated an object detector and an image classifier used in a pipeline, as described in §3.2. Our standalone image classifiers were trained and tested on uncropped images, while the image classifiers in our pipeline setup were trained on images that were cropped to just contain the subassembly. When testing the pipeline, we ran the object detector on uncropped images. Then, we cropped the image around the bounding box returned by the object detector, and then ran the image classifier on this cropped image. Figure 6.4 shows examples of cropped and uncropped images. The results of these pipelines are presented in Table 6.3.

The accuracy of the best pipeline was better than the accuracy of the best standalone DNN for all of our applications. Faster R-CNN and Fast MPN-COV was the best pipeline for all of our applications except Stirling, where EfficientDet-Lite2 and Resnet 50 worked better. This highlights the need for WCA application developers to determine the models that work best for their specific application.



Figure 6.4: Images showing steps from the Stirling and Meccano tasks. Uncropped images are on the left and cropped images are on the right.

	Meccano	Stirling	Sanitizer	Toyplane
EfficientDet-Lite0 and Resnet 50	75.0%	85.1%	87.9%	69.8%
EfficientDet-Lite0 and Fast MPN-COV	82.0%	78.4%	79.3%	77.2%
EfficientDet-Lite1 and Resnet 50	74.6%	70.3%	87.7%	70.9%
EfficientDet-Lite1 and Fast MPN-COV	81.7%	66.6%	79.3%	77.7%
EfficientDet-Lite2 and Resnet 50	75.0%	<b>91.0%</b>	89.1%	70.1%
EfficientDet-Lite2 and Fast MPN-COV	81.5%	86.0%	80.6%	76.6%
Faster R-CNN and Fast MPN-COV	<b>84.5%</b>	80.9%	<b>92.9%</b>	<b>81.9%</b>

Table 6.3: Classification accuracy for pipelines. Pipelines consist of an object detector, followed by an image classifier. Accuracy is the percentage of images that the model classified correctly. The highest accuracy for each application is in bold.

Device	Google Glass Enterprise Edition 2	Magic Leap 2	Vuzix Blade 2
Year Launched	2019	2022	2022
Weight	51 g	260 g	93 g
Computing Hardware	Qualcomm Snapdragon XR1	AMD Zen 2 and AMD RDNA 2	Quad Core ARM CPU
External Compute Pack	No	Yes	No
Spatial Mapping	No	Yes	No

Table 6.4: The smart glasses that we profiled our applications on. The values in the “Computing Hardware” row came from the tech specs advertised by the device manufacturer.

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
EfficientDet-Lite0 and Resnet 50	480 ± 14	161 ± 3	2031 ± 27
EfficientDet-Lite1 and Resnet 50	661 ± 46	183 ± 7	2423 ± 155
EfficientDet-Lite2 and Resnet 50	958 ± 9	222 ± 3	3072 ± 73

Table 6.5: Inference time for one frame, in milliseconds. For each cell, the average comes before the ± sign and the standard deviation comes after.

## 6.2.2 On-device WCA

The EfficientDet [83] object detector and Resnet 50 [49] image classifier can be run on certain Android devices using TensorFlow Lite. This allows developers to create WCA applications that run some or all of their computations on mobile devices instead of cloudlets. However, Fast MPN-COV and Faster R-CNN cannot currently run on mobile devices [11, 12].

We developed versions of our applications that ran EfficientDet and Resnet 50 pipelines directly on mobile devices. We profiled our applications on the three devices listed in Table 6.4. All three devices run Android, which allowed us to re-use our code across all of the devices.

### Inference Time

We first measured the amount of time it took to process an image through the two-DNN pipeline. We accomplished this by storing our test set on the devices, and running code that looped through each image. Inside the loop, our code ran the pipeline of models that was being timed. The code logged the elapsed time every 20 frames, based on Android’s uptime counter. Each pipeline was run for five minutes. Table 6.5 lists all of these times.

Chen et al. [30] conducted user studies to determine acceptable latency bounds for WCA applications. They found tight and loose latency bounds, which they describe as follows:

“The tight bound represents an ideal target, below which the user is insensitive to improvements, as measured, for example, by impact on performance or ratings of satisfaction. Above the loose bound, the user becomes aware of slowness, and user experience and performance is significantly impacted.”

The tight latency bound for an assembly application was 600 ms, and the loose bound was 2700 ms. Table 6.6 lists the largest pipeline that meets these latency bounds on each device.

	Tight Bound	Loose Bound
Google Glass EE 2	EfficientDet-Lite0 and Resnet 50	EfficientDet-Lite2 and Resnet 50
Magic Leap 2	EfficientDet-Lite2 and Resnet 50	EfficientDet-Lite2 and Resnet 50
Vuzix Blade 2	None	EfficientDet-Lite1 and Resnet 50

Table 6.6: The largest pipeline that meets tight and loose latency bounds. “Largest” refers to the number of parameters used for the pipeline’s version of EfficientDet. The latency bounds were determined by Chen et al. [30]. The accuracy of these pipelines is listed in Table 6.3.

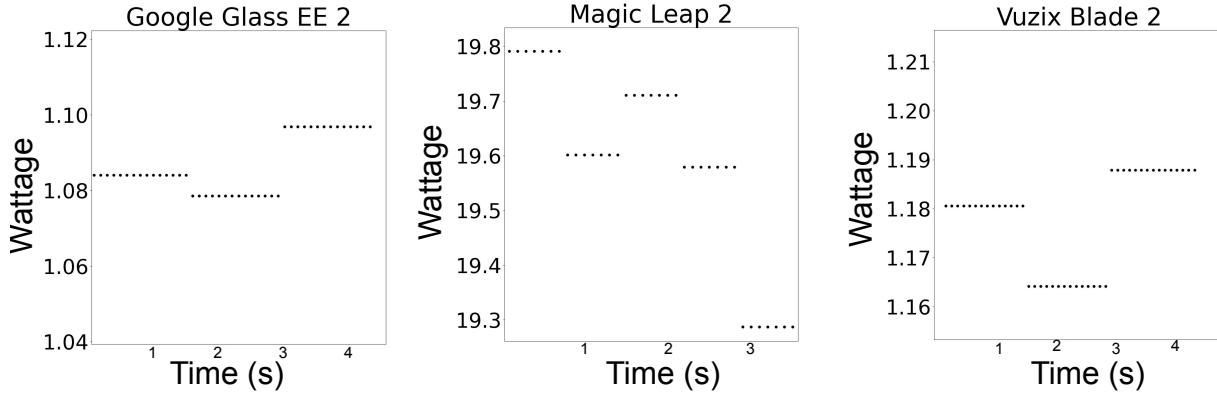


Figure 6.5: Power consumption for headsets running EfficientDet inference in a loop. The y-axes of these graphs do not start at zero. These values were sampled every 100 ms. Each dot on the graph represents one sample. The repeated values indicate that the current and voltage readings that Android provides access to are updated less frequently than 100 ms. As we describe in Section 6.2.2, the Magic Leap consumed significantly more power than the other devices that we tested.

## Power Consumption

We measured the amount of power that each of these devices used while running the pipelines in a loop. As with our previous experiments, we ran each pipeline for five minutes. None of these devices had user serviceable batteries, so we could not measure power consumption based on the current and voltage that was being supplied to the device by its charger. Instead, we ran our code with the devices unplugged, and queried for current and voltage readings from Android, using the `BatteryManager` class. We multiplied the voltage and current to compute power. Our code contained a background thread which logged the current and voltage every 100 ms. Figure 6.5 shows graphs of power for all three devices, sampled every 100 ms. Each value repeats several times, which indicates that current and voltage values were updated less frequently than 100 ms. Unfortunately, this means that these measurements are somewhat crude.

Table 6.7 lists the power values for each pipeline, running on all three devices. The *baseline* measurements were recorded for an application that showed an empty Android activity, but did not do anything aside from recording current and voltage values in a background thread. Table 6.8 lists the percentage increase in power consumption above the baseline, for running each

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
Baseline	$0.61 \pm 0.13$	$15.1 \pm 0.46$	$0.84 \pm 0.15$
EfficientDet-Lite0 and Resnet 50	$1.43 \pm 0.35$	$18.54 \pm 0.37$	$1.18 \pm 0.10$
EfficientDet-Lite1 and Resnet 50	$1.26 \pm 0.29$	$18.41 \pm 0.22$	$1.24 \pm 0.16$
EfficientDet-Lite2 and Resnet 50	$1.26 \pm 0.27$	$18.55 \pm 0.16$	$1.22 \pm 0.13$

Table 6.7: Average power consumption, in Watts. For each cell, the average comes before the  $\pm$  sign and the standard deviation comes after. These measurements were recorded while the mobile device was running the full pipeline. The baseline application did not carry out any computation.

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
EfficientDet-Lite0 and Resnet 50	134%	23%	41%
EfficientDet-Lite1 and Resnet 50	106%	22%	48%
EfficientDet-Lite2 and Resnet 50	107%	23%	45%

Table 6.8: The percentage increase of power consumption, above the baseline

pipeline. Percentage increase was calculated using the formula:

$$\frac{\text{Pipeline Wattage} - \text{Baseline Wattage}}{\text{Baseline Wattage}}$$

The Magic Leap 2 consumed over 15 watts running the baseline application. The device’s depth sensors might have consumed some of this power, or it could have been spatial mapping code running in the background. None of the pipelines increased the Magic Leap 2’s power usage by more than 25% of the power consumed by the baseline. The most dramatic increase over baseline power usage was for EfficientDet-Lite0 and Resnet 50 on Google Glass EE 2, with an average power usage of 1.43 Watts. However, this still implies a reasonable battery life. A 3.2 Wh battery can supply 1.43 Watts for over two hours.

Our takeaway message is that the reduction in battery life is not a reason for WCA application developers to be dissuaded from running DNNs on mobile devices. The decision should be made strictly based on accuracy. The models that can be run on mobile devices were significantly less accurate than the models that required cloudlets, for three out of the four applications that we tested. WCA application developers will have to test the accuracy of models trained on data for their specific applications, in order to determine if on-device models will provide adequate accuracy.

### 6.2.3 Split Computing

Split Computing offers a middle ground between offloading all computations to a server, and carrying out all computations locally. Instead, split computing uses a lightweight “head,” that runs lightweight computations on the mobile device, and a heavyweight “tail,” that runs the remainder of the computations on a more powerful server. This reduces the amount of data that



must be transmitted over the wireless network, and it reduces the amount of computation that is carried out on the cloudlet.

A large body of work exists exploring split computing applications. Odyssey [69] modified the Janus speech recognition application to operate in one of three modes. In one of the modes, a preliminary phase of speech processing was done locally (i.e., the head), and the extracted information was shipped to a remote server for the completion of the recognition process (i.e., the tail). For certain combinations of network bandwidth and device/server capabilities, this split offered lower end-to-end latency than fully local or fully remote execution. Several subsequent efforts in split computing [25, 37, 38, 68, 44, 92, 70, 26, 59] were surveyed in 2012 [39].

More recently, machine learning researchers have examined how DNN models can be split across mobile devices and servers [56, 53, 36, 64]. These works partition the DNNs such that the output from the subset of the network that runs on the mobile device (the head) is smaller than the original input to the network. We will henceforth refer to the output from the head as an embedding. Transmitting the embedding to the server, instead of the original input saves bandwidth. The embedding is a compressed representation of the input, that the remainder of the network (the tail) can process accurately. However, there is no clear way to convert an embedding back to the original input. A 2022 survey [65] discusses a number of recent works on split computing for DNNs. Unfortunately, modifying the architecture of a DNN for split computing is difficult [63]. Split architectures exist for common computer vision tasks like object detection and image classification. However, many developers lack the skills to create split architectures for tasks that such architectures don't already exist for.

Implementing the pipeline described in §3.2 using a split object detector is impractical. The output from a split object detector just contains the bounding box coordinates and class labels for detected objects. There is no way for the server to obtain a cropped image from the original embedding that was sent to the cloudlet. Our application would either have to send the entire image to the cloudlet along with the embedding, or it would have to send the bounding box coordinates back to the mobile device, and have the mobile device send the cropped image in some form to run the classifier. The former approach eliminates all of the bandwidth savings that split computing offers; while the latter approach requires a second round trip to the mobile device, which increases latency.

We instead realized split computing by running an EfficientDet object detector on a mobile device, and a Fast MPN-COV classifier on a cloudlet. The client that runs on the mobile device crops images around the bounding box with the highest confidence score, so only the detected subassembly remains in the image. This cropped image is then sent to the cloudlet, instead of the uncropped original. If the object detector does not find a subassembly in the image, the client does not send anything to the cloudlet. This setup is depicted in Figure 6.6.

## **Bandwidth Savings**

We compared the bandwidth required to transmit all images in full and the bandwidth required to transmit the cropped images from our split computing strategy. For each of the four applications, we ran the EfficientDet-Lite0 object detector that was trained for this application on the full test set. We recorded the number of bytes required to transmit the images cropped around the subassemblies detected by the model, and compared this with the number of bytes required

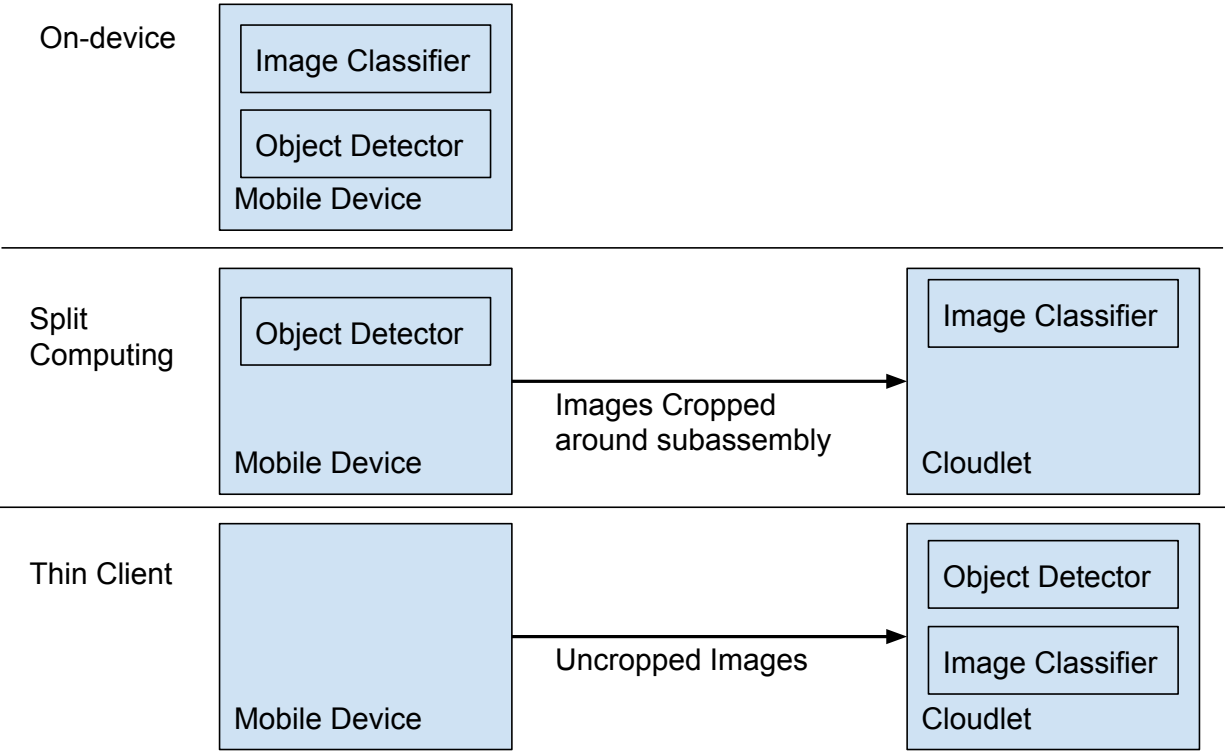


Figure 6.6: On-device, split computing, and thin client implementations of WCA. The split computing client only attempts to send an image when the object detector finds a subassembly. The thin client sends images as rapidly as the Gabriel flow control mechanism allows.

Stirling	Meccano	Toyplane	Sanitizer
79.2%	52.3%	86.8%	94.2%

Table 6.9: The bandwidth saved by transmitting cropped images. Images were cropped around the bounding boxes returned by EfficientDet-Lite0.

to transmit the entire test set. Table 6.9 lists the bandwidth savings percentages, which were calculated using the formula:

$$\frac{\text{Bytes for full images} - \text{Bytes for cropped images}}{\text{Bytes for full images}}$$

The bandwidth savings achieved by this strategy is content-dependant. The distance between the camera and the object being assembled will change how large a subassembly appears in the image, and this will directly impact the number of bytes required to transmit the cropped image. The bandwidth savings of techniques such as image compression or DNN-based split computing vary less based on the specific content in an image.

Transmitting cropped images required less than 50% of the bandwidth than the uncropped images would have required, for all of our datasets. The savings was over 90% for the Sanitizer dataset. In many cases, this significant bandwidth savings will be worth the reduction in accuracy (presented in Table 6.15).

### Inference Time

We measured the inference time of our split computing pipelines with code that ran the pipelines in a loop, similar to our measurements in Section 6.2.2. The loop runs through the images in our test set, and processes each one locally with an object detector. After an image is processed by the object detector, it is cropped if the detector finds a subassembly with high confidence. The crop is then sent to the cloudlet, where it is processed by the classifier. After sending a cropped image to the cloudlet, the mobile device immediately starts processing the next image. However, we used a single token for Gabriel’s flow control. The mobile client cannot send a new cropped image to the cloudlet before the cloudlet finishes processing the previous cropped image. In cases where the client is ready with the next cropped image before the cloudlet has finished processing the last one that was sent, the client will wait for the cloudlet to finish processing the last image before the client sends the next one.

The mobile devices were connected to the internet over Wi-Fi while the cloudlet had a wired connection to the internet. Figure 6.7 shows this topology. The ping time between the mobile device and the cloudlet was under 5 ms. The cloudlet had an Intel® Xeon® Processor E5–2699 CPU and an Nvidia GeForce GTX 1080 Ti GPU.

As with our other measurements, we ran each pipeline and device combination for five minutes. We logged the elapsed time every 20 iterations of the loop, and then divided the elapsed time by 20 to compute the per-frame inference time. Table 6.10 lists the averages and standard deviations of the per-frame inference times. Table 6.11 lists the largest split pipeline that meets the latency bounds from Chen et al. [30] on each device. Table 6.12 lists the accuracies of the pipelines from Tables 6.10 and 6.11.

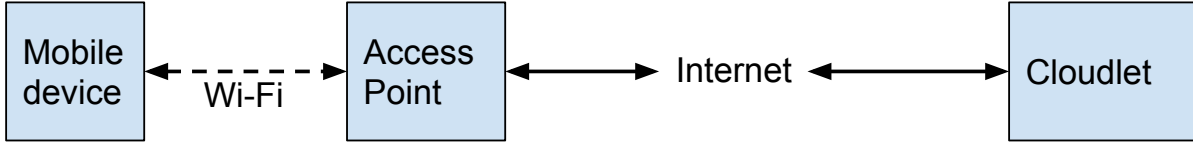


Figure 6.7: The network connecting mobile devices to the cloudlet. Solid lines represent a wired connection. The mobile device and cloudlet are in close network proximity to each other.

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
EfficientDet-Lite0 and Fast MPN-COV	308 ± 22	106 ± 7	1120 ± 47
EfficientDet-Lite1 and Fast MPN-COV	622 ± 186	133 ± 6	1778 ± 210
EfficientDet-Lite2 and Fast MPN-COV	779 ± 38	154 ± 4	2156 ± 65

Table 6.10: Single-frame inference time of split computing pipelines. The EfficientDet models were run on the device, then crops were sent to the cloudlet, where they were processed by Fast MPN-COV. These values include the DNN processing times on the device and cloudlet, as well as the time to send cropped images to the cloudlet and send results back to the mobile device. For each cell, the average comes before the ± sign and the standard deviation comes after.

	Tight Bound	Loose Bound
Google Glass EE 2	EfficientDet-Lite0 and Fast MPN-COV	EfficientDet-Lite2 and Resnet 50
Magic Leap 2	EfficientDet-Lite2 and Resnet 50	EfficientDet-Lite2 and Resnet 50
Vuzix Blade 2	None	EfficientDet-Lite2 and Resnet 50

Table 6.11: The largest split pipelines that meet tight and loose latency bounds. “Largest” refers to the number of parameters used for the pipeline’s version of EfficientDet. The latency bounds were determined by Chen et al. [30].

	Meccano	Stirling	Sanitizer	Toyplane
EfficientDet-Lite0 and Fast MPN-COV	82.0%	78.4%	79.3%	77.2%
EfficientDet-Lite1 and Fast MPN-COV	81.7%	66.6%	79.3%	77.7%
EfficientDet-Lite2 and Resnet 50	75.0%	91.0%	89.1%	70.1%
EfficientDet-Lite2 and Fast MPN-COV	81.5%	86.0%	80.6%	76.6%

Table 6.12: Accuracy of pipelines from Tables 6.10 and 6.11. This table is a subset of Table 6.3.

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
Baseline	$0.61 \pm 0.13$	$15.1 \pm 0.46$	$0.84 \pm 0.15$
EfficientDet-Lite0 and Fast MPN-COV	$1.37 \pm 0.11$	$18.03 \pm 0.32$	$1.27 \pm 0.10$
EfficientDet-Lite1 and Fast MPN-COV	$1.23 \pm 0.15$	$18.31 \pm 0.21$	$1.22 \pm 0.10$
EfficientDet-Lite2 and Fast MPN-COV	$1.21 \pm 0.16$	$18.76 \pm 0.25$	$1.24 \pm 0.12$

Table 6.13: Average power consumption of mobile devices running DNN pipelines. Power consumption is measured in Watts. For each cell, the average comes before the  $\pm$  sign and the standard deviation comes after. These measurements were recorded while the mobile device ran the object detector and then sent cropped images to the cloudlet, which then ran the image classifier. The baseline application did not carry out any computation.

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
EfficientDet-Lite0 and Fast MPN-COV	125%	19%	51%
EfficientDet-Lite1 and Fast MPN-COV	102%	21%	45%
EfficientDet-Lite2 and Fast MPN-COV	98%	24%	48%

Table 6.14: The percentage increase of power consumption, above the baseline

## Power Consumption

Similar to Section 6.2.2, we measured power consumption using a background thread that obtained current and voltage values from Android every 100 ms. Our power measurements were just made on the mobile devices, and the power consumed by the cloudlet was not measured. Table 6.13 lists the averages and standard deviations of our power measurements, while Table 6.14 lists the percentage increase over the baseline.

These values are similar to the power values for the pipelines that ran entirely on mobile devices. Split computing does not offer a significant reduction in power consumption compared to running DNNs entirely on mobile devices. Our discussion about power values from Section 6.2.2 applies to these values as well. Split computing consumes more power on the device than the thin clients discussed in Section 6.2.4.

## 6.2.4 Thin Clients

This section compares our on-device implementations of WCA applications with split computing implementations, and a thin client that runs both DNNs on a cloudlet.

### Accuracy

Table 6.15 lists the accuracies for the thin client and the highest accuracies for on-device and split computing pipelines. The thin client achieved the best performance for all applications except for Stirling. The best performing pipeline for Stirling can be run in all three configurations.

	Meccano	Stirling	Sanitizer	Toyplane
Best on-device pipeline	75.0%	91.0%	89.1%	70.9%
Best split computing pipeline	82.0%	91.0%	89.1%	77.7%
Thin client	84.5%	91.0%	92.9%	81.9%

Table 6.15: Classification accuracy for pipelines. Pipelines consist of an object detector followed by an image classifier. Accuracy is the percentage of images that the model classified correctly.

Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
166 ± 8	150 ± 8	203 ± 9

Table 6.16: Single-frame inference time for the thin client. Time is measured in milliseconds. For each cell, the average comes before the ± sign and the standard deviation comes after.

### Inference Time

Table 6.16 lists the inference times for all three devices running the thin client for five minutes. These times include transmitting images to the cloudlet, processing them there, and then transmitting results back to the mobile device. As with our other time measurements, the applications were run for five minutes, and elapsed time was recorded every 20 frames. These values were well below the tight latency bounds on all three devices.

The Vuzix Blade 2 could not meet the tight latency bound when running any of the split computing or on-device pipelines that we tested. The thin client was the only way we were able to meet the tight latency bound with the Vuzix Blade 2. However, the Google Glass EE 2 and Magic Leap 2 were able to meet the tight latency bound with some of the split computing and on-device pipelines that we tested.

### Power Consumption

Power measurements for the thin client, averaged over five minute runs, are listed in Table 6.17 along with the baseline measurements and on-device measurements from Section 6.2.2, and split computing measurements from Section 6.2.4. The baseline application showed an empty Android activity and just recorded current and voltage values in a background thread. Running the thin client on the Vuzix Blade 2 consumed more power than running the baseline application, but less power than running any of the on-device or split computing pipelines. The Google Glass EE 2 consumed slightly more power running the thin client as it did when running the on-device and split computing pipelines. However, all three of these clients consumed significantly more power than the baseline. The thin client on the Magic Leap 2 consumed slightly less power than the baseline application. There isn't a clear explanation for why this would have happened, but the difference is fairly small.

	Google Glass EE 2	Magic Leap 2	Vuzix Blade 2
Baseline	$0.61 \pm 0.13$	$15.1 \pm 0.46$	$0.84 \pm 0.15$
On-device	$1.26 \pm 0.27$	$18.41 \pm 0.22$	$1.18 \pm 0.10$
Split computing	$1.21 \pm 0.16$	$18.03 \pm 0.32$	$1.22 \pm 0.10$
Thin client	$1.30 \pm 0.41$	$14.26 \pm 0.33$	$1.17 \pm 0.09$

Table 6.17: Average power consumption for thin clients, in Watts. For each cell, the average comes before the  $\pm$  sign and the standard deviation comes after. The baseline application did not carry out any computation on the device, and it did not have any network communications. The on-device measurements are the smallest non-baseline value from each column in Table 6.7, while the split computing measurements are the smallest non-baseline values from Table 6.13. The thin client measurements were recorded while the mobile device sent an image to the cloudlet, the cloudlet ran the two DNNs, and then sent the result back to the mobile device.

### 6.3 Summary

Our new implementation of Gabriel is designed to be maintainable and easy to use. In addition, it allows connections from multiple users at the same time, which was previously unsupported.

Certain DNNs can run on mobile devices. This enables WCA applications to run entirely on a mobile device. Another option is to split the computation across a mobile device and a cloudlet. These options reduce the amount of network bandwidth that the application consumes. We found DNNs that can run on certain mobile devices with acceptable latency. However, these models were often less accurate than the DNNs that require more powerful servers.

We achieved the best performance for three out of our four applications offloading all computations to a cloudlet. However, cloudlet resources and bandwidth are limited resources. Split computing offers a significant bandwidth savings, with a modest cost in accuracy. It is also possible to run WCA applications entirely on a mobile device, but this hurts accuracy significantly.





# Chapter 7

## Conclusion and Future Work

This chapter concludes the dissertation by summarizing our contributions and discussing open problems and future work that will further our goal of making WCA applications practical.

### 7.1 Contributions

We address the problem of *Scaling Up Wearable Cognitive Assistance for Assembly Tasks* that involve many parts. The thesis we validated is:

**Scaling up WCA to complex assembly tasks 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.**

We propose computer vision techniques that make it possible to detect when steps of these long tasks have been completed. We also demonstrate the feasibility of training computer vision models for WCA using synthetic images. To handle the combinatorial explosion in the number of errors states for a task, we support escalation to a human task expert, and we create tools to determine the number of experts required to keep queuing times reasonable. We then redesign and re-implement the Gabriel software framework for WCA applications, to improve scalability. These contributions mark an important step towards making WCA applications practical for real world assembly tasks. Developers need a way to support real world assembly tasks, without spending an unreasonable amount of time collecting and labeling training images. WCA applications need some way to guide users who have made errors completing tasks. Lastly, WCA deployments must scale to multiple users.

## **7.2 Future Work**

### **7.2.1 Subassembly Identification**

WCA application developers need to split assembly tasks up into subassemblies. This is presently a manual process, but splitting tasks up automatically would reduce the burden on the developer. Gong et al. [43] automatically identified subassemblies of assembled objects based on CAD files. However, they were just concerned with making it as easy as possible for people to assemble the objects. We also need to detect completed task steps using computer vision. There are many ways to split an object up into different subassemblies. The split that results in the easiest assembly process might make the computer vision task difficult. Subassembly identification for WCA requires making the assembly process easy, and making the computer vision problem tractable. On top of this, WCA developers might not have access to the CAD model for a kit. WCA developers need a way to split objects into subassemblies without using a CAD model.

### **7.2.2 Detecting Environmental Issues**

Our computer vision models might output an incorrect label if the object being assembled is positioned at an awkward angle, or if the lighting in a room is too dim. WCA applications have no way of detecting when one of these issues has occurred. However, if the applications could detect issues like this, they could alert the user. The user could then correct the issue, and avoid the incorrect computer vision results that the issue would have caused. For example, the application could provide audio guidance to the user that says “hold this part differently, or at a different angle.”

Practitioners running WCA applications can record traces of real users completing a task with the help of a WCA application. This will allow the practitioners to collect images with objects at an angle that causes the application’s models to output an incorrect result. The practitioners can then train the model to recognize an error state that corresponds to an instruction to correct the issue.

### **7.2.3 Computer Vision Techniques**

Deep Neural Networks perform best when the training, validation, and testing data are drawn from the same distribution as the data that the trained model will be used on [74]. Unfortunately, this will rarely be the case in practice for WCA applications. The lighting conditions that the application is used in and defects in how parts are manufactured can introduce biases in the data that the application will see at runtime. Biases that did not exist in the training data can cause the DNNs used by the application to perform poorly. Each task that a developer creates a WCA application for requires its own training set. This limits the size of the training set that would be practical to collect in order to develop one of these applications. Computer vision models that are robust to biases that did not exist in the original training set, or models that can be refined at runtime to address biases, will improve the reliability of WCA applications.

The computer vision models used by our applications are trained to identify the completed step shown in an image. But they are not trained on any data depicting incomplete steps. Our

applications currently avoid telling a user that a step has been completed, while the user is actually in the middle of the step, using the techniques described in §3.4. In addition, users hands often cover up parts of an assembly while they are completing steps of a task, which prevents the object detector from finding the subassembly in an image. However, additional work is needed to reliably prevent the application from giving the next instruction while the user is still in the middle of the previous step of the task.

#### **7.2.4 Textures for 3D Models**

CAD designs for born-digital objects specify the shapes of parts. However, they rarely include information about the materials that objects are manufactured from. In fact, a single CAD design can be used to manufacture objects out of multiple different materials. A person who wants to generate synthetic data for a new born-digital object must specify texture information for the material that the object was made out of. This is a time consuming process. In particular, the person must ensure that the object looks realistic in a variety of simulated lighting conditions. Reducing the manual effort required to specify texture information will make it easier to generate synthetic training images.

#### **7.2.5 Device and Cloudlet Implementations**

Wearable devices that are more powerful than the ones used in our experiments are likely to be developed. With any mobile device, there is a tension between the device's size and weight, its computing capability, and its battery life. As mobile devices improve, devices without such constraints are likely to improve more [80]. For this reason, we foresee that there will continue to be a gap between the accuracy of models that can be run directly on mobile devices, and models that can be run on a cloudlet. WCA applications will thus still require edge computing in order to achieve the highest possible accuracy for computer vision models. However, these applications will benefit from improvements to future smart glasses.

In addition to hardware improvements, new computer vision techniques will be developed that will further push the boundaries on computer vision processing that can take place on mobile devices. WCA applications can leverage these improvements through thin clients and split computing approaches similar to those detailed in Chapter 6. These applications can also run local computations to avoid sending certain frames to the cloudlet. For example, an early discard filter might determine that there is no way that a frame shows that the current step has been completed, so the frame will not require any further processing. If this filter can be run on a mobile device, the device can avoid transmitting filtered frames to the cloudlet.

#### **7.2.6 Development Tools**

Open Workflow Editor can only be used to create applications that give the user instructions and then process camera images until the application determines that the user has completed a step or reached an error state. Certain task steps might be time consuming, which makes it computationally expensive to process frames for the entire time a user is completing a step. Allowing developers to limit periods when images are being processed would save on these costs.

For example, a user could press a button on the side of the headset, in order to indicate that they believe that a step has been completed. The application would then only have to process images after this button is pressed. Open Workflow Editor will have to be extended in order to support this.

Applications created with Open Workflow Editor must determine when steps have been completed based on the output of DNNs. However, developers might want to employ other techniques, such as length measurement or rules-based classifiers. In addition, Open Workflow Editor only supports applications that process images from an RGB camera. Allowing developers to create applications with Open Workflow Editor that utilize techniques that are not DNNs and sensors that are not RGB cameras will require a substantial software engineering effort. Almost every WCA application that has been developed for assembly tasks exclusively utilizes RGB cameras and DNNs for image processing. Any developer considering adding a new feature to Open Workflow Editor should first develop new WCA applications that use this feature, without using Open Workflow Editor. This will establish the value of such a feature, and help the developer understand how it should be implemented. This experience will also inform the user interface that Open Workflow Editor should have developers use to set up this feature in new applications.

The tools used for synthetic training image generation require a CAD design file for the object being assembled. Other strategies must be employed to make it feasible for developers to create WCA applications for kits that the developers do not have CAD designs for. These strategies might involve generating synthetic images based on 3D scans, data augmentation based on photographs of the kit, or training computer vision models using few-shot learning [88].

## **7.2.7 Multi-Modal Sensing**

This dissertation examines WCA applications that determine when task steps have been completed by processing images from an RGB camera. Collecting data from torque sensors and force sensors would increase what applications can detect, beyond what is possible with RGB cameras alone. For example, an application could determine if a user tightened a bolt too tightly, or if a user did not attach two parts together with enough force. Sensors beyond RGB cameras might be of particular use for the assembly of objects that are extremely large. Determining step completion for objects like these might require a user to stand back, so that an entire object is in view of a headset's camera. The work of Antifakos et al. [22] represents one extreme, where an entire Ikea wardrobe was outfitted with gyroscopes, accelerometers, force sensing resistors, and infrared distance meters in order to determine step completion. However, multi-modal sensing with a mix of these sensors and an RGB camera might represent the best of both worlds. An RGB camera can be used where possible, to avoid having to install sensors on the kit being assembled. But torque sensors and force sensors can be used for steps whose completion cannot be sufficiently detected using RGB cameras.

# Bibliography

- [1] Computer vision annotation tool: A universal approach to data annotation. <https://www.intel.com/content/www/us/en/developer/articles/technical/computer-vision-annotation-tool-a-universal-approach-to-data-annotation.html>.
- [2] Gabriel android client. <https://repo1.maven.org/maven2/edu/cmu/cs/gabriel/>.
- [3] Camerax overview. <https://developer.android.com/training/camerax>.
- [4] Dynamics 365 remote assist. <https://dynamics.microsoft.com/en-us/mixed-reality/remote-assist/>.
- [5] The gabriel framework for wearable cognitive assistance using cloudlets. <https://github.com/cmusatyalab/gabriel>.
- [6] Lamp assistant. <https://github.com/cmusatyalab/gabriel-ikea>.
- [7] Lego assistant. <https://github.com/cmusatyalab/gabriel-lego>.
- [8] Gabriel python client. <https://pypi.org/project/gabriel-client/>.
- [9] Sandwich assistant. <https://github.com/cmusatyalab/gabriel-sandwich>.
- [10] Gabriel server. <https://pypi.org/project/gabriel-server/>.
- [11] Running tf2 detection api models on mobile. [https://github.com/tensorflow/models/blob/master/research/object\\_detection/g3doc/running\\_on\\_mobile\\_tf2.md](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/running_on_mobile_tf2.md).
- [12] Autodiff for user script functions. <https://github.com/pytorch/pytorch/issues/22329>.
- [13] Vieaura. <https://vieaura.com/>.
- [14] Webex expert on demand. <https://www.webex.com/industries/frontline.html>.
- [15] Openworkflow. <https://github.com/cmusatyalab/OpenWorkflow>.
- [16] The websocket protocol. <https://tools.ietf.org/html/rfc6455>.
- [17] Ama xperteye. <https://www.amaxperteye.com/>.
- [18] Zeromq. <https://zeromq.org/>.

- [19] Zoom client sdks. <https://marketplace.zoom.us/docs/sdk/native-sdks/>.
- [20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [21] 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.
- [22] 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.
- [23] Autodesk. Assembly hierarchy in the browser. <https://knowledge.autodesk.com/support/inventor-products/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/Inventor-Help/files/GUID-9C1118A3-7EEB-4479-AEF7-BCA37F9F907F-htm.html>.
- [24] 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>.
- [25] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang. The Case for Cyber Foraging. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.
- [26] R. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying Cyber Foraging for Mobile Devices. In *Proceedings of the 5th International Conference on Mobile Systems Applications and Services*, San Juan, Puerto Rico, June 2007.
- [27] Steve Borkman, Adam Crespi, Saurav Dhakad, Sujoy Ganguly, Jonathan Hogins, You-Cyuan Jhang, Mohsen Kamalzadeh, Bowen Li, Steven Leal, Pete Parisi, Cesar Romero, Wesley Smith, Alex Thaman, Samuel Warren, and Nupur Yadav. Unity perception: Generate synthetic data for computer vision. *CoRR*, abs/2107.04259, 2021. URL <https://arxiv.org/abs/2107.04259>.
- [28] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [29] Lawrence Brown, Noah Gans, Avishai Mandelbaum, Anat Sakov, Haipeng Shen, Sergey

- Zeltyn, and Linda Zhao. Statistical analysis of a telephone call center: A queueing-science perspective. *Journal of the American Statistical Association*, 100(469):36–50, 2005. ISSN 01621459. URL <http://www.jstor.org/stable/27590517>.
- [30] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiyong 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>.
- [31] Intergraph Corporation. Learn about the assembly hierarchy. <https://docs.hexagonppm.com/r/en-US/Intergraph-Smart-3D-Planning/Version-2016-11.0/60340>.
- [32] Michael RW Dawson. Fitting the ex-gaussian equation to reaction time distributions. *Behavior Research Methods, Instruments, & Computers*, 20(1):54–57, 1988.
- [33] 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.
- [34] 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.
- [35] Boris Epshtein and Shimon Ullman. Semantic hierarchies for recognizing objects and parts. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007. doi: 10.1109/CVPR.2007.383086.
- [36] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services. In *Proceedings of the 2019 IEEE/ACM Int. Symposium on Low Power Electronics and Design (ISLPED)*, 2019.
- [37] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.
- [38] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy Conservation and Application Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.
- [39] Jason Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing via Opportunistic Offload*. Morgan & Claypool Publishers, 2012.
- [40] 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.

- [41] 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>.
- [42] Timnit Gebru, Jonathan Krause, Yilun Wang, Duyun Chen, Jia Deng, and Li Fei-Fei. Fine-grained car detection for visual census estimation. *Proceedings of the AAAI Conference on Artificial Intelligence*, Feb. 2017. URL <https://ojs.aaai.org/index.php/AAAI/article/view/11174>.
- [43] Hanqing Gong, Lingling Shi, Dongmei Liu, Jiahui Qian, and Zhijing Zhang. Construction and implementation of extraction rules for assembly hierarchy information of a product based on ontostep. *Procedia CIRP*, 97:514–519, 2021. ISSN 2212-8271. doi: <https://doi.org/10.1016/j.procir.2020.08.008>. URL <https://www.sciencedirect.com/science/article/pii/S2212827120315006>. 8th CIRP Conference of Assembly Technology and Systems.
- [44] S. Goyal and J. Carter. A Lightweight Secure Cyber Foraging Infrastructure for Resource-constrained Devices. In *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications*, 2004.
- [45] Stefano Gualandi and Giuseppe Toscani. Call center service times are lognormal: a fokker-planck description. *Mathematical Models and Methods in Applied Sciences*, 28(08):1513–1527, 2018.
- [46] 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>.
- [47] 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.
- [48] 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>.
- [49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [50] 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.

- [51] 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>.
- [52] Stefan Hinterstoisser, Olivier Pauly, Hauke Heibel, Marek Martina, and Martin Bokeloh. An annotation saved is an annotation earned: Using fully synthetic training for object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.
- [53] Ke-Jou Hsu, Ketan Bhardwaj, and Ada Gavrilovska. Couper: DNN Model Slicing for Visual Analytics Containers at the Edge. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, Arlington, VA, 2019.
- [54] 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>.
- [55] 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.
- [56] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS '17*, Xi'an, China, April 2017.
- [57] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953. ISSN 00034851. URL <http://www.jstor.org/stable/2236285>.
- [58] 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.
- [59] Mads Darø Kristensen. Execution Plans for Cyber Foraging. In *MobMid '08: Proceedings of the 1st Workshop on Mobile Middleware*, Leuven, Belgium, 2008.
- [60] 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>.
- [61] 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.
- [62] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollar, and Larry Zitnick. Microsoft coco: Common objects in context. In *ECCV. European Conference on Computer Vision*, September 2014. URL <https://www.microsoft.com/en-us/research/publication/microsoft-coco-common-objects-in-context/>.
- [63] Yoshitomo Matsubara and Marco Levorato. Split computing for complex object detectors: Challenges and preliminary results. In *Proceedings of the 4th International Workshop on Embedded and Mobile Deep Learning, EMDL'20*, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380737. doi: 10.1145/3410338.3412338. URL <https://doi.org/10.1145/3410338.3412338>.
- [64] Yoshitomo Matsubara, Sabur Baidya, Davide Callegaro, Marco Levorato, and Sameer Singh. Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems. In *Proceedings of the 2019 MobiCom Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019.
- [65] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges. *ACM Computing Surveys*, March 2022.
- [66] Julien Miranda., Stanislas Larnier., Ariane Herbulot., and Michel Devy. Uav-based inspection 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.
- [67] 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>.
- [68] Dushyanth Narayanan and Mahadev Satyanarayanan. Predictive Resource Management for Wearable Computing. In *Proceedings of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.
- [69] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.
- [70] MinHwan Ok, Ja-Won Seo, and Myong-soon Park. A Distributed Resource Furnishing to Offload Resource-Constrained Devices in Cyber Foraging Toward Pervasive Computing. In Tomoya Enokido, Leonard Barolli, and Makoto Takizawa, editors, *Network-Based Information Systems*, volume 4658 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

- [71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [72] 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>.
- [73] 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>.
- [74] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5389–5400. PMLR, 09–15 Jun 2019. URL <https://proceedings.mlr.press/v97/recht19a.html>.
- [75] 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>.
- [76] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.
- [77] Mahadev Satyanarayanan. The Emergence of Edge Computing. *IEEE Computer*, 50(1), 2017.
- [78] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), 2009.
- [79] 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.mobibase.2014.257757.
- [80] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. The role of edge offload for hardware-accelerated mobile devices. In *Proceedings of the 22nd*

*International Workshop on Mobile Computing Systems and Applications*, HotMobile '21, page 22–29, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383233. doi: 10.1145/3446382.3448360. URL <https://doi.org/10.1145/3446382.3448360>.

- [81] 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](https://doi.org/10.1007/978-1-4899-0718-9_31).
- [82] S.Maragatha Sundari and Santhanagopalan Srinivasan. M/m/c queueing model for waiting time of customers in bank sectors. *International Journal of Mathematical Sciences & Applications*, 1, 01 2011.
- [83] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- [84] 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.
- [85] 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>.
- [86] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.
- [87] 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>.
- [88] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Comput. Surv.*, 53(3), jun 2020. ISSN 0360-0300. doi: 10.1145/3386252. URL <https://doi.org/10.1145/3386252>.
- [89] Ye Wang, Norman Mu, Daniele Grandi, Nicolas Savva, and Jacob Steinhardt. A3d: Studying pretrained representations with programmable datasets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*,

pages 4878–4889, June 2022.

- [90] Junhui Wu, Yun Ye, Yu Chen, and Zhi Weng. Spot the difference by object detection. *CoRR*, abs/1801.01051, 2018. URL <http://arxiv.org/abs/1801.01051>.
- [91] Hui Xiong, Lu Ma, Mengxi Ning, Xu Zhao, and Jinxian Weng. The tolerable waiting time: A generalized pareto distribution model with empirical investigation. *Computers & Industrial Engineering*, 137:106019, 2019. ISSN 0360-8352. doi: <https://doi.org/10.1016/j.cie.2019.106019>. URL <https://www.sciencedirect.com/science/article/pii/S0360835219304772>.
- [92] Su Ya-Yunn and Jason Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *Proceedings of the 3rd International Conference on Mobile systems, Applications, and Services*, 2005.