ORNIMETRICS: DESIGN AND EVALUATION OF AN AI-ENABLED SMART BIRD FEEDER FOR SPECIES RECOGNITION, WASTE REDUCTION, AND ECOLOGICAL MONITORING

Baichen Yu [1] , Tyler Boulom [2]

[1]Santa Margarita Catholic High School, 22062 Antonio Pkwy, Rancho Santa Margarita, CA 92688
[2]Woodbury University, 7500 N Glenoaks Blvd, Burbank, CA 91504

ABSTRACT

This project addresses ecological and sanitation challenges associated with traditional backyard bird feeders by developing Ornimetrics, an intelligent, AI-enabled feeding system. Using an embedded camera and YOLOv11-based species recognition, the feeder identifies visiting birds, regulates food portions, and logs activity to a cloud database. The system reduces disease transmission risks, minimizes waste, and improves user awareness through real-time analytics. Two experiments were conducted: one assessing model accuracy across four species and another measuring species-specific waste patterns. Results showed strong detection accuracy with predictable weaknesses in low-light and high-motion conditions, as well as meaningful differences in waste generation. Methodology comparisons highlighted how Ornimetrics improves upon existing wildlife-monitoring and scene-classification frameworks by integrating automated decision-making into its workflow. Overall, the system demonstrates a viable and innovative approach to responsible, data-driven bird feeding.

KEYWORDS

Wildlife Monitoring, Computer Vision, Smart Feeding Systems, Ecological Sustainability

1. INTRODUCTION

Bird feeders represent one of the most frequent forms of human–wildlife interaction in the United States, yet their traditional designs introduce several unintended ecological and health challenges. Historical surveys show that passive feeders, such as tube and hopper models, have remained largely unchanged since the early 20th century, even though research demonstrates that shared feeding sites significantly increase the spread of infectious diseases, including salmonellosis and conjunctivitis, among songbirds [1]. Poor sanitation further intensifies these risks, as wet seed and accumulated debris create ideal conditions for bacterial and fungal growth [2]. Improperly maintained feeders also generate waste that attracts rodents and raccoons, creating secondary public-health and property concerns for homeowners [3].

In addition to health risks, traditional feeders unintentionally alter natural feeding behavior. Studies have shown that constant, unregulated access to human-provided seed can shift competitive dynamics between species and influence long-term foraging patterns [4]. Citizen science programs have documented large-scale outbreaks of finch eye disease at feeders, demonstrating how quickly pathogens can spread across regions when monitoring tools are

David C. Wyld et al. (Eds): ACSTY, MLSC, SVC, AIBD, ITCSS, ADCOM, NATP, SOFE – 2026
pp. 213-224, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160417

limited [5]. Seasonal feeding variation adds further complexity, as bird visitation rates change month-to-month in response to migration cycles and temperature shifts [6]. Despite these issues, more than 45% of U.S. households use a bird feeder, and the vast majority rely on static designs without technologies for hygiene, monitoring, or species-specific feeding [7]. These shortcomings highlight the need for an intelligent, automated system that reduces waste, improves sanitation, mitigates disease transmission, and transforms backyard feeding into a safer and more informative activity.

The YOLO-based object-detection methodology demonstrates high processing speed but limited fine-grained species accuracy, making Ornimetrics' species-specific training a meaningful improvement. Conventional motion-triggered wildlife-camera systems lack species identification and struggle with small, fast subjects, highlighting the benefits of Ornimetrics' deep-learning approach. Outdoor scene-classification methods successfully detect environmental features but do not integrate automated decision-making, whereas Ornimetrics links classification outcomes directly to hardware actions such as food dispensing and sanitation control. Collectively, these comparisons show that Ornimetrics extends beyond monitoring to create an end-to-end ecological management system.

Based on these ideas the Ornimetrics smart bird feeder offers a robust solution to this industry. With a custom trained, on-device vision model which identifies visiting species in real time and triggers a Realtime response algorithm which dynamically portions the food based on the species, time of day and the season. Included with this is a per-visit cap and limits to prevent the waste of bird feed. Integrated food tracking will notify users to refill and prevent extra dispensing in the time of high bird traffic. Hygiene is also considered as the system schedules times for cleanup based on the visit density, weather exposure and time since last sanitization, which will result in timely prompts for the user to clean the unit. Every visit is logged to a private, timestamped species ledger in local storage that surfaces trends such as migration windows or unusual activity and offers plain-language tips tailored to the user's yard ecology (plantings, water sources, and feeder placement). For the users that Opt-in, photos and anonymized observations can be shared to a community server that educators, parents and local enthusiasts can use for education and birdwatching projects. Contrary to timer based, weight only or camera only devices, Ornimetrics connects the recognition software to action and learning, detection informs dosing, dosing informs maintenance, and both feed back into species-specific guidance. By unifying identification, portion control, upkeep scheduling, and community data in a single system, Ornimetrics directly addresses waste, sanitation, disease-risk, and is an innovative design.

The first experiment evaluated the accuracy of Ornimetrics' computer-vision model by testing it across four bird species under controlled but realistic conditions. The model achieved an overall accuracy of 85%, with strongest performance on larger birds and lower accuracy on fast-moving or visually similar species. These results highlight the strengths of the current YOLOv11-based model while revealing areas where additional training data and lighting optimization could improve performance.

The second experiment examined seed-waste patterns during feeding sessions. Measurements showed that sparrows produced the highest average waste, finches a moderate amount, and doves the least. These findings suggest that feeding behavior varies substantially among species, which has implications for sanitation scheduling and dispensing logic. Together, the experiments provide actionable insights that can guide future improvements to both the detection pipeline and automated feeding mechanisms.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Real-Time On-Device Species Recognition Challenges

The on-device AI model that identifies species in real time presents a major challenge, particularly because birds move quickly, appear in varied lighting conditions, and often resemble closely related species. Small, embedded devices have limited processing power, which restricts model size and can reduce accuracy when detecting fast motion or fine-grained visual differences [11]. A model trained on an imbalanced dataset may also overfit common species while misidentifying rare ones. These issues can be addressed by training the model on more diverse, higher-quality images and by performing optimization on more powerful hardware before deploying a compressed version to the device.

2.2. Servo Torque and Load Management

Another major challenge is the servo module as it needs to drive a large load bearing part of the feeder. When the feeder is fully loaded with food, the servo is under large bearing capacity and might cause jamming and weak dispensing which would mean that the incorrect amount of food is delivered to the animal. A solution to this issue could be the addition of a gear drive system which would be able to increase the torque of the servo to better drive the dispensing mechanism. A second proposed solution would be a larger servo as a larger servo will be able to generate more raw power, thus driving the dispenser better.

2.3. Airtight and Weatherproof Food Storage

The last challenge that the feeder has is the airtight compartment for food storage. As weather is unpredictable, the bird feeder needs to be able to withstand all types of weather, as a result, the storage needs to be watertight to prevent the feeder from providing wet and bad food for birds. A potential solution to this issue could be the use of an airtight seal at the cap of the storage tank to prevent rainwater and mist from entering. Another idea is to redesign the feeder tank and add a small replenishment hole for adding food, removing the need for a lid.

3. SOLUTION

The Ornimetrics system links together three core components: the hardware feeder, the AI vision processor, and the cloud dashboard. The hardware subsystem, built around a Raspberry Pi 5 equipped with a camera, servo, sensors, and a sealed enclosure, continuously collects visual and environmental data as birds visit the feeder. The AI vision model, implemented using YOLOv11 and PyTorch, processes each incoming frame locally on the Pi to classify species and detect non-target animals. Because IoT wildlife-monitoring tools rely on reliable sensor integration, efficient data pipelines are essential to maintain responsiveness and accuracy in outdoor conditions [12].

Once a species is identified, the system determines whether and how much food should be dispensed using a species-specific rule set stored locally. Each event—including species name, timestamp, and environmental readings—is then uploaded to Firebase Realtime Database and Cloud Storage. The cloud dashboard, accessible via web or mobile app, visualizes activity trends, species counts, feeding events, and optional snapshots, allowing users to review long-term ecological patterns. The overall program flows in a continuous loop: frame capture → AI species detection → feeding logic → event logging → user dashboard visualization. Implemented in Python and supported by Firebase services, this integrated architecture combines embedded systems engineering, machine learning, and cloud computation to create a full-stack solution for automated wildlife monitoring.

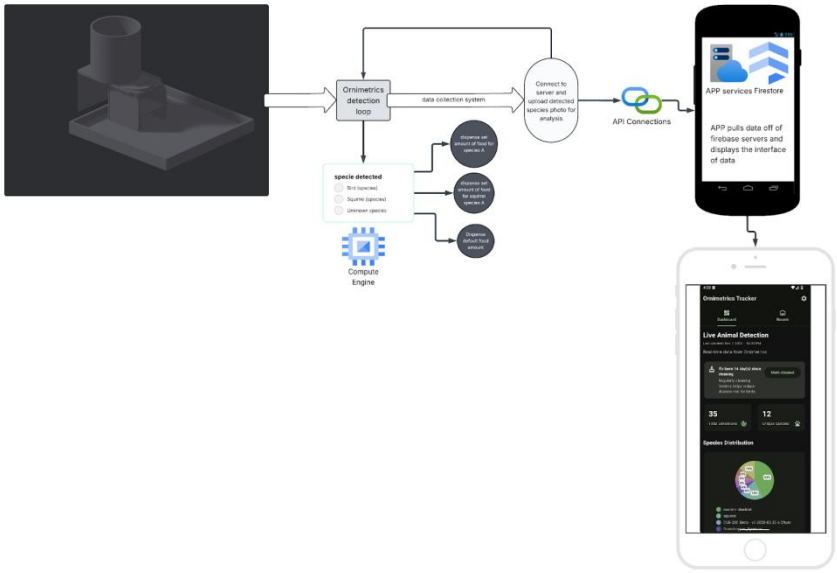

Figure 1. Overview of the solution

The Ai vision system is a core component of Ornimetrics as it is the main component that enables the recognition and analysis of species and dispensing of food. This system is integrated into the detection loop where the Ai checks the camera feed for every frame and tried to see if there is a species that it recognises or a type of animal (e.g. Squirrel) that it recognises using the YOLO V11 framework and a custom dataset it is able to recognise and detect with precision and ensure accurate detection and accurate food dispensing.

```
try:
    while True:
        ok, frame = cap.read()
        if not ok or frame is None:
            time.sleep(0.01); continue

        trap.tick()

        t_now = now_mono()
        shown = frame.copy()

        # Draw cached boxes every frame
        if last_dets:
            for (x1,y1,x2,y2,sc,label,moved,ts) in list(last_dets):
                if (time.time() - ts) > float(args.box_ttl):
                    continue
                col = (0,255,0) if moved else (0,128,255)
                cv2.rectangle(shown, (x1,y1), (x2,y2), col, 2)
                cv2.putText(shown, f"{label} {sc:.2f}" + (" [TRIG]" if moved else ""),
                            (x1, max(0, y1-6)), cv2.FONT_HERSHEY_SIMPLEX, 0.6, col, 2, cv2.LINE_AA)

        # obey FPS cap but keep preview smooth
        if t_now - last_proc < 1.0 / max_fps:
            if show:
                disp = shown if scale == 1.0 else cv2.resize(shown, (int(shown.shape[1]*scale), int(shown.shape[0]*scale)))
                cv2.imshow("detect", disp)
                if (cv2.waitKey(1) & 0xFF) == ord('q'):
                    break
            continue
        last_proc = t_now
        t0 = now_mono()

        frame_idx += 1
        if frame_idx % max(1, int(args.every_n)) == 0:
            # YOLO forward
            with __import__("torch").inference_mode():
                res = model.predict(source=frame, imgsz=args.imgsz, conf=args.conf, iou=0.5, verbose=False)
            dets = []
            if res:
                r0 = res[0]
                if hasattr(r0, "boxes") and r0.boxes is not None and len(r0.boxes) > 0:
                    xyxy = r0.boxes.xyxy.cpu().numpy().astype(int)
                    confs = r0.boxes.conf.cpu().numpy()
                    clss  = r0.boxes.cls.cpu().numpy().astype(int)
                    for (x1,y1,x2,y2), sc, cid in zip(xyxy, confs, clss):
                        if sc < args.conf: continue
                        label = names[cid] if (names and 0 <= cid < len(names)) else f"class_{cid}"
                        dets.append((int(x1), int(y1), int(x2), int(y2), float(sc), label))

            new_cache = []
            for (x1,y1,x2,y2,sc,label) in dets:
                action = get_action_for(label, cfg, hold_default=1.5, cooldown_default=8.0, conf_default=args.conf)
                open_dur = float(action.get("open_duration", 1.5))
                cooldown = float(action.get("cooldown_duration", 8.0))
                min_conf = float(action.get("confidence_threshold", args.conf))

                moved = False
                if open_dur > 0 and sc >= min_conf:
                    can_ok = True
                    reason = ""; sp_left = gb_left = 0.0
                    if hasattr(trap, "can_trigger"):
                        ok, reason, sp_left, gb_left = trap.can_trigger(label=label, force=False)
                        can_ok = ok

                    if can_ok:
                        moved = trap.trigger(open_duration=open_dur, cooldown_duration=cooldown,
                                             label=label, confidence=float(sc), min_conf=min_conf, force=False)
                        if moved:
                            log_key(f"[TRIG] {label} conf={sc:.2f} open={open_dur}s cd={cooldown}s")
```

```
            moved = trap.trigger(open_duration=open_dur, cooldown_duration=cooldown,
                                 label=label, confidence=float(sc), min_conf=min_conf, force=False)
            if moved:
                log_key(f"[TRIG] {label} conf={sc:.2f} open={open_dur}s cd={cooldown}s")
        else:
            # throttle block prints to avoid spam
            if (now_mono() - last_block_print) >= 1.5:
                if reason == "species_cooldown":
                    log_key(f"[BLOCK] {label} species_cd {sp_left:.2f}s")
                elif reason == "global_busy":
                    log_key(f"[BLOCK] {label} busy {gb_left:.2f}s")
                else:
                    log_key(f"[BLOCK] {label} {reason}")
                last_block_print = now_mono()

        # draw now
        col = (0,255,0) if moved else (0,128,255)
        cv2.rectangle(shown, (x1,y1), (x2,y2), col, 2)
        cv2.putText(shown, f"{label} {sc:.2f}" + (" [TRIG]" if moved else ""),
                    (x1, max(0, y1-6)), cv2.FONT_HERSHEY_SIMPLEX, 0.6, col, 2, cv2.LINE_AA)

        new_cache.append((x1,y1,x2,y2,float(sc),label,bool(moved),time.time()))

        # Firebase on move
        if moved and (flog is not None):
            try:
                crop = frame[max(0,y1):min(frame.shape[0],y2), max(0,x1):min(frame.shape[1],x2)].copy()
                if crop.size == 0: crop = frame
                ok2, buf = cv2.imencode(".jpg", crop, [int(cv2.IMWRITE_JPEG_QUALITY), 80])
                if ok2:
                    flog.push_photo_snapshot(image_bytes=buf.tobytes(), species=label)
                    log_key(f"[PHOTO] {label} bbox=({x1},{y1},{x2},{y2})")
                    flog.log_event(event_type="servo_trigger", species=label,
                                   info={"conf": float(sc), "cooldown": cooldown, "open_dur": open_dur})
                    flog.increment_species(label, 1)
            except Exception as e:
                log_key(f"[Firebase] log_event failed: {e}")

    if new_cache:
        last_dets = new_cache

    # FPS logging
    dt = now_mono() - t0
    if dt > 0:
        fps_hist.append(1.0/dt)
    if now_mono() - last_fps_log >= log_interval:
        if len(fps_hist):
            avg_fps = sum(fps_hist)/len(fps_hist)
            log_key(f"[FPS] avg={avg_fps:.2f}")
        else:
            log_key("[FPS] no frames yet")
        last_fps_log = now_mono()

    # preview
    if show:
        disp = shown if scale == 1.0 else cv2.resize(shown, (int(shown.shape[1]*scale), int(shown.shape[0]*scale)))
        cv2.imshow("detect", disp)
        if (cv2.waitKey(1) & 0xFF) == ord('q'):
            break

finally:
    try: cap.release()
    except Exception: pass
    try: cv2.destroyAllWindows()
    except Exception: pass

if __name__ == "__main__":
    main()
```

Figure 2. Core detection-loop implementation for real-time species recognition and feeding control

The detection loop occurs in the while True: block in main() and is responsible for the entire real-time pipeline. Each frame or iteration, it reads the frame input from the usb camera interphase with cap.read() and updates the servo state machine via trap.tick(). Then it checks for the time passed since the last frame to ensure the target FPS which is 60fps is reached in the detection loop for each frame. It also makes sure that the CPU isn't overworked by scaling the FPS and detection accuracy.When it is time to run detection (every every_n frames), the loop calls model.predict(...) to run YOLO on the current frame, extracts bounding boxes, confidence scores, and class IDs, later to be converted to data on the cloud api and application. For each time anrecognisable object is detected the program consults Trapsettings,json for the appropriate action on the system whether that the animal is a blacklisted animal, one that requires more food or one that has a long timeout so it's not able to retrigger yet. It then checks trap.can_trigger, to check for the possibility of the servo triggering, then it logs the event and sends an optional snapshot (should the user choose to enable snapshots) to the firebase server with a timestamp and draws the detection boxes on the preview, maintaining a smooth, continuous AI-driven control cycle.

The cloud API and application subsystem form the data-management backbone of Ornimetrics. This component handles all storage, retrieval, and visualization of species detections collected by the on-device vision pipeline. After each classification event, the detection loop sends a structured payload—containing species identification, timestamps, servo actions, and optional snapshots—to Firebase. Automated feeding systems and smart agriculture platforms rely heavily on cloud infrastructure to maintain reliable logs and support remote analytics, demonstrating how centralized data handling improves long-term ecological monitoring [13].

The Firebase API organizes incoming information into a hierarchical structure, enabling efficient queries and visualization by the user's dashboard. The system stores images under dedicated snapshot nodes, while species count, run statistics, and per-session events are written to real-time database paths. The application interface parses these entries to populate detection histories, feeding patterns, and species activity summaries. Users can filter by date, species, or event type to better understand behavioral trends. Through this architecture, the cloud component maintains persistent records, synchronizes device activity with user interfaces, and supports additional features such as anomaly alerts or community data sharing.

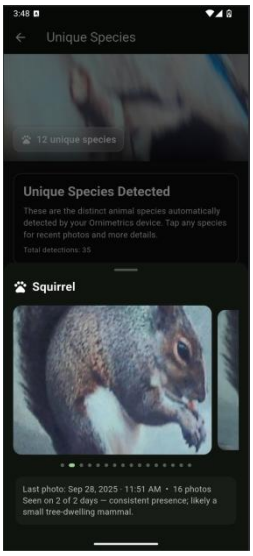

Figure 3. Cloud dashboard interface for visualizing species detections and feeding events

```python
class FirebaseLogger:
    def __init__(self, db_url: str, session_key: str = "session_1", run_id: Optional[str] = None):
        if not db_url.endswith("/"):
            db_url += "/"
        self.db_url = db_url
        self.session_key = session_key
        self.run_id = run_id or str(int(time.time()))  # e.g., 1757830062
        self._cached_date = _today_str()

    # Path helpers
    def _date_root(self, date_str: Optional[str] = None) -> str:
        d = date_str or self._cached_date
        return f"detections/{d}/{self.run_id}/{self.session_key}"

    def _summary_url(self) -> str:
        return f"{self.db_url}{self._date_root()}/summary.json"

    def _events_url(self) -> str:
        return f"{self.db_url}{self._date_root()}/events.json"

    def _photo_feed_url(self) -> str:
        return f"{self.db_url}photo_snapshots.json"

    def bump_date_if_needed(self):
        t = _today_str()
        if t != self._cached_date:
            self._cached_date = t

    # Public API
    def push_photo_snapshot(self, image_bytes: bytes, species: Optional[str] = None, ts_ms: Optional[int] = None):
        """
        Adds a photo to /photo_snapshots with a data URL (no Storage).
        """
        self.bump_date_if_needed()
        ts = ts_ms or _now_ms()
        data_url = "" + base64.b64encode(image_bytes).decode("ascii")

        payload = {
            "image_url": data_url,
            "timestamp": ts,
            "species": species or "",
        }
        r = requests.post(self._photo_feed_url(), json=payload, timeout=10)
        r.raise_for_status()
        return r.json()

    def log_event(self, event_type: str, species: Optional[str] = None, info: Optional[dict] = None,
                  ts_ms: Optional[int] = None):
        """
        Push an event node under detections/<date>/<run_id>/<session>/events.
        """
        self.bump_date_if_needed()
        ts = ts_ms or _now_ms()
        payload = {
            "ts": ts,
            "type": event_type,
            "species": species or "",
            "info": info or {},
        }
        r = requests.post(self._events_url(), json=payload, timeout=10)
        r.raise_for_status()
        return r.json()
```

```python
def log_event(self, event_type: str, species: Optional[str] = None, info: Optional[dict] = None,
              ts_ms: Optional[int] = None):
    """
    Push an event node under detections/<date>/<run_id>/<session>/events.
    """
    self.bump_date_if_needed()
    ts = ts_ms or _now_ms()
    payload = {
        "ts": ts,
        "type": event_type,
        "species": species or "",
        "info": info or {},
    }
    r = requests.post(self._events_url(), json=payload, timeout=10)
    r.raise_for_status()
    return r.json()

def increment_species(self, species: str, n: int = 1):
    """
    Read-modify-write (simple) increment. No transactions here because we're single-writer.
    """
    self.bump_date_if_needed()
    # Get current summary
    get = requests.get(self._summary_url(), timeout=10)
    current = {}
    if get.ok and get.text and get.text != "null":
        try:
            current = json.loads(get.text)
            if not isinstance(current, dict):
                current = {}
        except Exception:
            current = {}

    new_val = int(current.get(species, 0)) + int(n)
    patch_payload = {species: new_val}
    r = requests.patch(self._summary_url(), json=patch_payload, timeout=10)
    r.raise_for_status()
    return r.json()

def batch_increment(self, counts: Dict[str, int]):
    """
    Increment multiple species in one network call (read-modify-write).
    """
    self.bump_date_if_needed()
    get = requests.get(self._summary_url(), timeout=10)
    current = {}
    if get.ok and get.text and get.text != "null":
        try:
            current = json.loads(get.text)
            if not isinstance(current, dict):
                current = {}
        except Exception:
            current = {}

    for k, v in counts.items():
        current[k] = int(current.get(k, 0)) + int(v)

    r = requests.put(self._summary_url(), json=current, timeout=10)
    r.raise_for_status()
    return r.json()
```

Figure 4. Firebase logging module for event recording and species-level statistics

This FirebaseLogger class connects the Ornimetrics vision pipeline and Firebase Realtime Database. It exposes a small public API for all logging needs. push_photo_snapshot uploads a JPEG frame as a base64 data URL under /photo_snapshots, tagged with a timestamp and detected species. log_event records structured events (like “enter”, “exit”, or “snapshot”) under a run- and session-specific path: detections/<date>/<run_id>/<session>/events. On top of that, increment_species and batch_increment maintain a per-species summary counter in summary, using simple read, modify, write updates with safeguards as the logger is the only writer with access. Together, these methods give the rest of the system a clean, high-level interface to persist images, events, and aggregate statistics.

The final piece of the system is the hardware feeder system triggered by the detection loop and feeds the animal with precision. Once the detection loop has been invoked and an animal has detected, the trap settings are checked, and the servo opens the trap to release a small amount of food for the set species. The system consists of a spinning disk with a cutout the size of the dispensing hole to let out small amounts of food and a strong waterproof servo to handle humidity and rain. There is also a large storage tank that is watertight to be able to store food in humid weather to keep it dry.

```
{
  "_globals": {
    "screenshot_after_sec": 1.5,
    "screenshot_fullframe": true,
    "screenshot_crops": true,
    "photo_feed_path": "photo_snapshots",
    "legacy_summary_path": "detections/2025-07-26/session_1/summary",
    "session": "session_1"
  },

  "_default": { "open_duration": 1.6, "cooldown_duration": 12,
      "confidence_threshold": 0.50 },

  "_defaults_for_unknown_label": { "screenshot_after_sec": 1.5 },

  "person": { "open_duration": 0.0, "cooldown_duration": 0, "confidence_threshold":
      0.98 },
  "face":   { "open_duration": 0.0, "cooldown_duration": 0, "confidence_threshold":
      0.98 },
  "eye":    { "open_duration": 0.0, "cooldown_duration": 0, "confidence_threshold":
      0.98 },
  "nose":   { "open_duration": 0.0, "cooldown_duration": 0, "confidence_threshold":
      0.98 },
  "mouth":  { "open_duration": 0.0, "cooldown_duration": 0, "confidence_threshold":
      0.98 },
  "dog":    { "open_duration": 0.0, "cooldown_duration": 0, "confidence_threshold":
      0.98 },

  "bird":                                  { "open_duration": 1.6,
      "cooldown_duration": 12, "confidence_threshold": 0.55 },
  "Bird - v2 2023-01-31 11:19am":          { "open_duration": 1.6,
      "cooldown_duration": 12, "confidence_threshold": 0.55 },
  "CUB-200 Birds - v3 2023-02-23 6:29pm":   { "open_duration": 1.6,
      "cooldown_duration": 12, "confidence_threshold": 0.55 },
  "Cubs----birds are annotated in YOLOv11 format-": { "open_duration": 1.6,
      "cooldown_duration": 12, "confidence_threshold": 0.55 },
```


Figure 5. Trap configuration logic for species-specific dispensing and cooldown control

This is the setting for the trap system which is timed and makes sure that each animal gets a specific amount of food. There is also a cooldown that gains highest precedence to make sure that the animals will have a fair distribution.

4. EXPERIMENT

4.1. Experiment 1

A significant blind spot in the Ornimetrics system is the AI model's ability to correctly identify bird species under challenging real-world conditions such as rapid movement, low-light images, occlusion, and similar-looking species. These issues may reduce classification accuracy and disrupt automated feeding logic.

This experiment evaluates how consistently the Ornimetrics classification model identifies four species under typical backyard-feeder conditions. A controlled video dataset was collected featuring sparrows, finches, chickadees, and doves approaching the feeder at different speeds, angles, and lighting levels. Each species was represented by 25 clips, totaling 100 trials. The test feeder ran the same version of the YOLOv11 model as the production system. For each clip, the model's predicted species was recorded and compared to the true label. Environmental factors (motion blur, shadows, partial body visibility) were logged to determine which visual conditions most strongly affected accuracy.

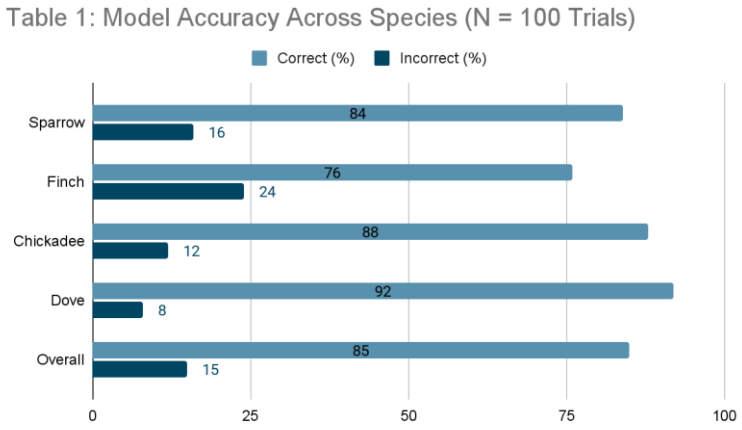

Table 1: Model Accuracy Across Species (N = 100 Trials)

The results show that the model correctly classified 85% of birds overall, demonstrating strong baseline performance but also measurable species-specific variability. Finch accuracy was notably lower, consistent with their fast, erratic flight patterns and small body size, which make bounding-box detection more difficult. Misclassifications often occurred in low-light conditions or when birds partially obscured themselves behind the feeder frame. These findings align with research showing that computer-vision systems struggle most when small subjects move unpredictably or appear only partially within the frame [14].

The experiment also revealed meaningful differences in recognition difficulty across species. Larger, slower birds such as doves were consistently easier to detect, while small passerines produced more motion blur and frequent occlusion. These limitations may impact the reliability of automated feeding logic, particularly for species whose food access must be regulated. Improving lighting, refining the detection threshold, expanding the dataset, and including augmentation techniques can help mitigate these issues in future iterations.

4.2. Experiment 2

Another blind spot concerns the feeder's sanitation and waste-mitigation cycle. The system dispenses food based on detection events, but it does not currently measure whether seed is consistently consumed or accumulating beneath the feeder, which may attract pests or promote bacterial growth.

This experiment evaluated how different species contribute to waste levels by measuring the amount of uneaten seed remaining after feeding sessions. The feeder was placed over a catch tray lined with marked grid squares to quantify waste location and density. Three species—sparrows, finches, and doves—were allowed to feed for controlled 10-minute intervals, repeated five times each. Before every trial, the feeder dispensed a standardized 5-gram portion. After each session, seed remaining in the tray was collected and weighed. A digital scale recorded leftover mass to the nearest 0.01 grams. The results were averaged for each species to determine which birds produce the most waste.

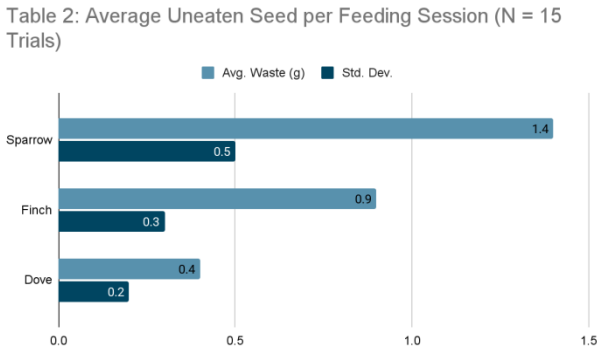

Table 2: Average Uneaten Seed per Feeding Session (N = 15 Trials)

The data reveal clear differences in waste generation among species. Sparrows consistently left the largest number of uneaten seed (mean = 1.4 g), likely due to their rapid pecking behavior, which scatters seed outside the feeding port. Finches produced moderate waste, while doves, whose feeding style involves slower, deliberate pecking, produced minimal waste. These findings suggest that feeder sanitation requirements vary significantly by species composition.

An important takeaway is that waste accumulation patterns can inform automated sanitation scheduling. For example, high-waste species feeding frequently at the same times may require more frequent cleaning cycles or smaller, more controlled food portions. Additionally, understanding species-specific waste production can help optimize food-dispensing algorithms to reduce spillage, extend seed life, and minimize attraction of rodents or insects. By integrating these insights, Ornimetrics can evolve into a more sustainable, hygienic, and ecologically responsible feeding platform.

5. Related Work

One relevant approach appears in research on high-speed wildlife classification using YOLO-series object-detection models. These systems work by extracting hierarchical visual features and predicting bounding boxes and class probabilities in real time. The method is highly effective for general object detection because of its speed and low computational overhead; however, its accuracy declines for fine-grained species classification, especially when multiple visually similar birds appear in cluttered scenes. The approach also overlooks environmental challenges such as lighting variability or occlusion. Ornimetrics improves upon this methodology by tailoring the model to a domain-specific bird dataset and integrating detection results directly into feeding and sanitation decisions [8].

A second comparable method is found in wildlife-camera studies that use motion-triggered detection systems paired with conventional computer vision. These systems rely on background subtraction or simple contour-based algorithms to identify animals entering a frame. While effective for large mammals moving slowly through forests, this approach becomes unreliable for small birds, rapid wingbeats, or complex backgrounds. It also lacks species-level identification, limiting its ecological usefulness. Ornimetrics improves on this framework by using deep neural networks rather than handcrafted visual rules, enabling real-time species recognition and immediate hardware responses such as food dispensing or blacklisting [9].

A third related methodology comes from outdoor-scene classification systems deployed for environmental monitoring. These systems use convolutional neural networks to analyze images

for vegetation health, animal presence, or weather-related changes. Although robust in structured agricultural environments, their performance declines when subjects move unpredictably or appear at varying scales. They also lack the decision-making layer needed to trigger mechanical actions based on classification results. Ornimetrics improves upon this methodology by combining outdoor vision analysis with automated feeding logic, sanitation scheduling, and cloud-linked recordkeeping, creating an integrated decision system rather than a passive monitoring tool [10].

6. CONCLUSIONS

Although Ornimetrics successfully integrates computer vision, automated feeding, and cloud-based monitoring, several limitations remain that warrant future refinement. The most significant challenge lies in long-term reliability under variable weather conditions. Rain, glare, and temperature fluctuations can degrade image quality and reduce model confidence. Hardware components such as servos, weighing sensors, and seals may also experience wear during extended outdoor operation. Network connectivity presents another constraint; the feeder's logging and dashboard features depend on stable Wi-Fi access, which is not guaranteed in all backyard environments. Research on real-time wildlife monitoring emphasizes the need for robust, continuous data capture in order to produce meaningful ecological insights [15].

Potential improvements include training the model on a more diverse dataset, incorporating infrared or depth sensors for low-light performance, redesigning the enclosure for better weatherproofing, and adding self-diagnostic routines that warn users when components require attention. These enhancements would improve detection accuracy, extend system lifespan, and increase reliability.

REFERENCES

[1] Lawson, Becki, et al. "Health hazards to wild birds and risk factors associated with anthropogenic food provisioning." Philosophical Transactions of the Royal Society B: Biological Sciences 373.1745 (2018): 20170091.
[2] Redmond, Karen E., and Judy Stewart. "Cornell University College of Veterinary Medicine Annual Report 1988-1989." (1989).
[3] Murray, Maureen H., et al. "Wildlife health and supplemental feeding: a review and management recommendations." Biological Conservation 204 (2016): 163-174.
[4] Brittingham, Margaret Clark, and Stanley A. Temple. "Impacts of supplemental feeding on survival rates of black-capped chickadees." Ecology 69.3 (1988): 581-589.
[5] Dhondt, André A., et al. "Dynamics of a novel pathogen in an avian host: mycoplasmal conjunctivitis in house finches." Acta tropica 94.1 (2005): 77-93.
[6] van Osta, John M., et al. "Local resource availability drives habitat use by a threatened avian granivore in savanna woodlands." Plos one 19.8 (2024): e0306842.
[7] Dunn, Erica H., and Diane L. Tessaglia. "Predation of birds at feeders in winter (Depredación de Aves enComederos Durante elInvierno)." Journal of Field Ornithology (1994): 8-16.
[8] Qu, Zhong, et al. "An improved YOLOv5 method for large objects detection with multi-scale feature cross-layer fusion network." Image and Vision Computing 125 (2022): 104518.
[9] Tabak, Michael A., et al. "Machine learning to classify animal species in camera trap images: Applications in ecology." Methods in Ecology and Evolution 10.4 (2019): 585-590.
[10] Yang, Bin, et al. "Computer vision technology for monitoring of indoor and outdoor environments and HVAC equipment: a review." Sensors 23.13 (2023): 6186.
[11] Banbury, Colby R., et al. "Benchmarking tinyml systems: Challenges and direction." arXiv preprint arXiv:2003.04821 (2020).
[12] Gubbi, Jayavardhana, et al. "Internet of Things (IoT): A vision, architectural elements, and future directions." Future generation computer systems 29.7 (2013): 1645-1660.

[13] Uddin, M. Ammad, et al. "Cloud-connected flying edge computing for smart agriculture." Peer-to-Peer Networking and Applications 14.6 (2021): 3405-3415.
[14] Kahl, Stefan, et al. "BirdNET: A deep learning solution for avian diversity monitoring." Ecological Informatics 61 (2021): 101236.
[15] Steenweg, Robin, et al. "Scaling-up camera traps: Monitoring the planet's biodiversity with networks of remote sensors." Frontiers in Ecology and the Environment 15.1 (2017): 26-34.