International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

OOP-Based WhatsApp Chatbot for Educational Use

SASWATH K R[1], DR MADHU MITHA K.[2]

[2]Assistant Professor
[1,2]Department of Computing Technologies, SRM Institute of Science and Technology

Abstract

The integration of conversational agents into ubiquitous messaging platforms enables interactive, always-available learning experiences. This paper presents the design and development of a rule-based WhatsApp chatbot that teaches Object-Oriented Programming (OOP) concepts while itself being implemented with OOP principles. The system uses Node.js (ES Modules) with Express for routing, MongoDB for persistent logs, and Redis for lightweight session state. WhatsApp connectivity is provided through Twilio's Messaging API; ngrok is used for secure webhook exposure in development. The architecture is modular, with classes for menu rendering, input parsing, session management, and intent routing. We demonstrate encapsulation, inheritance, polymorphism, and abstraction throughout the codebase and show that deterministic, menu-driven conversations are effective for beginner-friendly programming instruction. In tests on Twilio's sandbox, the bot achieved sub-250 ms median response time, stable concurrency for 15+ simulated users, and positive qualitative feedback. We discuss limitations and outline extensions including multilingual support and future NLP-based intent handling.

Keywords- WhatsApp chatbot, Object-Oriented Programming, Twilio, Node.js, Express, MongoDB, Redis, Educational technology, Rule-based dialog

1. INTRODUCTION

- Messaging applications are a natural medium for educational assistants due to their reach and familiarity. WhatsApp's global adoption enables learners to access help without new accounts, apps, or installations. However, many educational chatbots either rely on opaque NLP pipelines or lack pedagogical structure, especially for foundational topics such as OOP.
- We develop a WhatsApp chatbot that explains OOP concepts—classes, objects, encapsulation, inheritance, polymorphism, and abstraction—through short, deterministic responses and a guided menu flow. Crucially, the bot's internal design mirrors the concepts it teaches: class-based handlers encapsulate behaviors, a base handler interface enables polymorphic responses, and shared abstractions isolate infrastructure (Twilio, persistence) from pedagogy (topic content).

A. Problem Statement and Objectives

- Beginners often struggle to connect abstract OOP concepts to working software. Traditional

materials can be static and non-interactive. We therefore design a chatbot that (i) delivers concise, structured explanations of OOP topics over WhatsApp, (ii) embodies OOP in its internal architecture, (iii) integrates Twilio for reliable message delivery, and (iv) remains modular and extensible for future NLP upgrades.

B. Contributions

- A practical OOP-driven architecture for an educational, rule-based chatbot on WhatsApp.
- A clean separation of concerns via class-based handlers, a router, and services (Twilio, persistence, session), illustrating encapsulation, inheritance, polymorphism, and abstraction.
- An implementation using Node.js/Express, MongoDB, and Redis that achieves low-latency responses in sandbox, evaluations.
- A reproducible configuration workflow for Twilio WhatsApp sandbox and local development

2. RELATED WORK

Chatbots in education. Prior work shows chatbots can increase engagement by enabling on-demand, conversational explanations in domains from programming to general STEM. Deterministic, rule-based flows are attractive for formative instruction where consistency and assessment alignment matter.

WhatsApp and Twilio. WhatsApp does not expose a native DIY chatbot interface; Twilio bridges this via programmable messaging. Incoming WhatsApp messages hit a Twilio webhook that forwards payload to the application server; the server's response is relayed back to the user, enabling near real-time interaction.

Rule-based vs. NLP bots. Rule-based agents simplify validation and guarantee predictable outputs, while NLP bots (e.g., based on spaCy/BERT/Dialogflow) enable flexible phrasing and intent recognition at the cost of training, tuning, and occasional non-determinism. For introductory instruction, our system prioritizes control and transparency, while remaining extensible for future NLP upgrades.

3. PROBLEM DEFINITION

Learners new to object-oriented programming (OOP) often struggle to connect concise definitions with working mental models and everyday examples. Existing resources are either static (notes/videos that don't adapt) or open-ended AI chatbots that can be inconsistent and difficult to validate in instructional settings. At the same time, students overwhelmingly spend time on messaging apps, where lightweight, low-friction learning could happen—but WhatsApp lacks a native, pedagogy-first teaching flow:

Constraints & assumptions

- Platform constraint: WhatsApp messaging via Twilio webhooks (no custom WhatsApp UI).
- Deployment constraint: lightweight Node.js/Express service with simple data stores (e.g., Redis for session state, MongoDB for logs).
- Pedagogy constraint: deterministic, rule-based responses prioritized over free-form NLP.
- Privacy assumption: no storage of personally identifying content beyond minimal messaging metadata required for sessioning.

• Environment assumption: development accessible via secure public endpoint (e.g., tunnel) and configurable secrets.

Functional requirements

- Present main and sub-menus for OOP topics; accept numeric and keyword selections.
- Deliver concise definition + example per topic; offer **Next / Back / Menu** navigation.
- Handle unexpected inputs with clear guidance; allow users to re-enter the flow quickly.
- Persist and restore user session context (current topic/step).
- Record timestamped request/response logs.

Non-functional requirements (success criteria)

- Predictability: identical inputs in the same state yield identical outputs.
- Responsiveness: median end-to-end reply under ~250 ms in a development setup.
- Robustness: no crashes under at least ~15 concurrent user sessions; zero unhandled exceptions in routine tests.
- Maintainability: add new topics by subclassing/registration without modifying existing handlers (Open/Closed Principle).
- Security: secrets via environment variables; request validation in production; minimal data retention.

Out of scope (initial version)

- Free-form natural-language intent detection, code execution/grading, and multimedia lessons; these may be added in future iterations.

4. SYSTEM ARCHITECTURE

A. Technology Stack

The backend uses JavaScript (ES Modules) with Node.js and Express for HTTP endpoints, Twilio's API for WhatsApp integration, MongoDB (via Mongoose) for structured logging, Redis for in-memory session state, and ngrok for secure localhost tunneling during development [4]–[6].

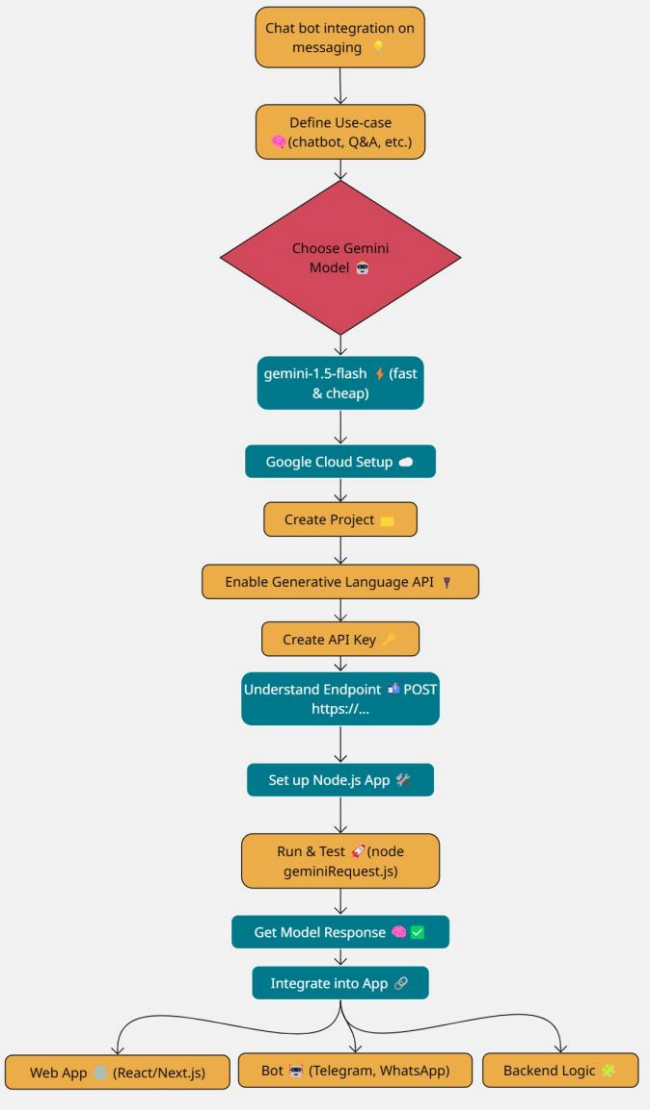

Fig. 1. Flowchart of Chatbot integration

B. High-Level Flow

When a user sends a WhatsApp message, Twilio invokes the bot's webhook with message metadata. The server normalizes input, routes it via an intent router to a corresponding handler class, and returns a templated reply. Logs are persisted to MongoDB; Redis caches recent session context (e.g., last menu) for continuity.

C. OOP Design

The design includes (a) a MessageRouter that dispatches to handlers, (b) a MenuManager that renders main/sub-menus, (c) a TwilioService that encapsulates messaging I/O, and (d) a SessionStore backed by Redis. A base Handler defines the interface (canHandle, handle); specialized handlers (e.g., InheritanceHandler, EncapsulationHandler) extend it.

- **Encapsulation**: services wrap external concerns (Twilio, DB, cache)..
- **Polymorphism**: router invokes handle on the matched handler at runtime.

- Abstraction: external modules interact through concise interfaces.

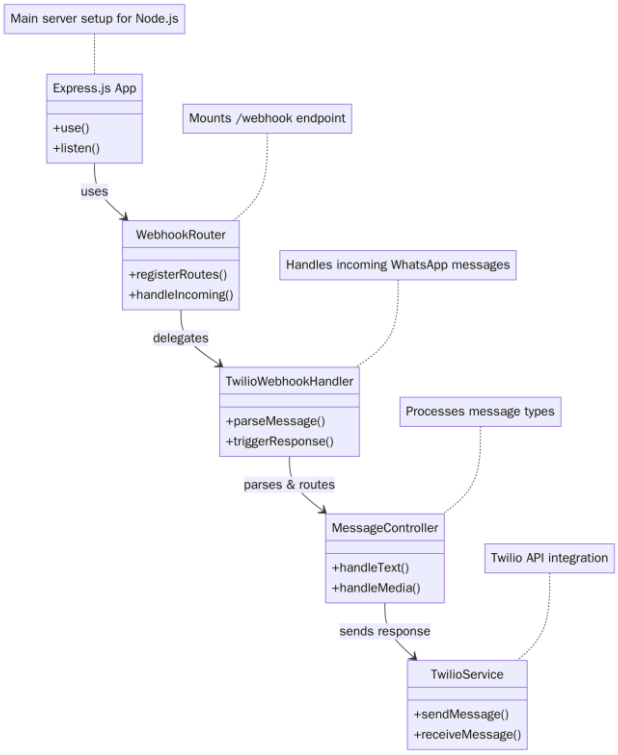

Fig. 2. UML Class Diagram of the OOP-Based Chatbot

5. METHODOLOGY

This section details how the WhatsApp OOP tutor was conceived, engineered, and evaluated. It covers requirements capture, architecture, message-flow logic, state management, content authoring, deployment, testing, and metrics—so you can reproduce or extend the system.

A. Requirements Elicitation and Scope

The preprocessing pipeline is an important part in providing consistency in the data and in increasing the efficiency of learning. Steps include:

- Pedagogical needs: short, deterministic explanations of OOP topics (classes/objects, encapsulation, inheritance, polymorphism, abstraction) with minimal cognitive load.
- User constraints: learn inside WhatsApp; tolerate typos; resume where they left off.
- System constraints: lightweight backend, simple persistence, fast responses, minimal PII retention.
- Success criteria: predictable outputs (same input → same response), median latency <250 ms (dev),

stable with ≥15 concurrent users.

B. System Architecture

- Runtime: Node.js (ES Modules) + Express.
- Messaging: WhatsApp via Twilio Programmable Messaging webhooks.
- State: Redis (per-user session/context).
- Persistence: MongoDB (Mongoose) for message logs and simple analytics.
- Config/Secrets: environment variables; centralized config module.
- Local tunneling (dev): ngrok for public HTTPS to webhook.

Core components (class-oriented):

- MessageHandler (abstract): interface with canHandle(state, input) and respond(context).
- Topic handlers: EncapsulationHandler, InheritanceHandler, PolymorphismHandler, AbstractionHandler, etc.
- MenuManager: renders menus, maps choices/keywords to handlers, manages "menu/back/help".
- Router: normalizes input, selects handler (polymorphism), orchestrates response.
- TwilioService: verifies requests (prod), parses inbound payloads, sends outbound messages.
- SessionService: get/set user state in Redis with TTL.
- DbService: asynchronous inserts for logs; backpressure-aware.
- ErrorMiddleware: centralized error capture, classification, and safe fallbacks.

C. Message Lifecycle (End-to-End)

- Inbound: WhatsApp → Twilio → HTTPS POST to /webhook/whatsapp.
- Normalization: lowercase, trim, collapse whitespace; extract keywords and numeric options.
- Context load: fetch user session (state machine node) from Redis.
- Routing: Router queries MenuManager + handlers via canHandle(...).
- Response compose: selected handler returns text (definition, mini-example, next options).
- State update: advance/retain state (e.g., Topic=Encapsulation, Step=Example).
- Persist: async log to MongoDB (user hash, input, output, timestamps, state snapshot).
- Outbound: TwilioService replies to WhatsApp; delivery status optionally recorded.
- Observability: structured log (JSON) with correlation id for tracing

D: Conversation State Machine

States (simplified):

● WELCOME → MAIN_MENU → TOPIC_MENU → TOPIC_EXPLAIN → TOPIC_EXAMPLE → (NEXT_TOPIC | TOPIC_MENU)

● **Global interrupts: HELP, MENU, BACK.**

Transitions:

● Numeric choice 1/2/3… or keyword (e.g., "polymorphism") advances deterministically.

● Invalid input → FALLBACK (one-line guidance + show valid actions).

● MENU jumps to MAIN_MENU; BACK returns to previous valid state.

Session policy:

● Redis key: sess:<user_hash>; fields: state, topic, step, updated_at.

● TTL (e.g., 24 h) to evict stale sessions and minimize data retention.

E. Content Authoring and Delivery

● Content blocks per topic: {definition, one-liner, micro-example, analogy, next-hint}.

● Authoring format: small JSON/TS objects co-located with handlers or loaded from a content/ directory.

● Determinism: each topic returns the same text for the same step; no randomization.

F. Input Handling and Intent Resolution

● Accept numbers (menu indices) and keywords (topic names/synonyms).

● Keyword map includes common misspellings (e.g., "poly morphism").

● Priority: if in TOPIC_MENU, keywords map to visible topics; otherwise prompt to type menu.

● Ambiguity: prefer current state's valid transitions; else show minimal disambiguation.

G. Error Handling and Fallbacks

● User errors: respond with a friendly single-line tip + valid options.

● System errors: generic apology + preserve state; log stack and classification (Twilio, Redis, Mongo,

App).

- Retries: limited retry on transient Redis/Mongo ops; circuit breaker to avoid cascading failures.
- Time-outs: graceful time-out handling with idempotent webhook processing.

H. Data Model and Persistence

MongoDB collections:

- message_logs: {_id, user_hash, direction, text, state_snapshot, ts}
- events (optional): health pings, delivery receipts.

Indexes: {user_hash, ts} for session traces; TTL index (optional) for retention policy.

Privacy: store a salted/hashed user identifier; no message content beyond instructional necessity if policy requires.

I. Security and Compliance

- Secrets via environment variables; no hard-coded tokens.
- Validate Twilio request signatures (production).
- HTTPS everywhere; restrict IPs if hosting allows.
- Least-privilege database users; parameterized queries.
- Data retention window aligned with institutional policy; document consent if storing transcripts.

J. Deployment Workflow

- Dev: Node server + ngrok URL → configure Twilio Sandbox webhook.
- Staging/Prod: public HTTPS (reverse proxy), environment-specific Redis/Mongo, autoscaling app container.
- Config bundles: .env.<env> with CI/CD to inject secrets.
- Observability: structured logs, minimal dashboard: requests/min, median/95p latency, error rate.

K. Testing Methodology

- Unit tests: handlers, router, session service, content selector.
- Integration tests: end-to-end webhook flows with mocked Twilio payloads.
- Load tests: synthetic users (≥15 concurrent), measure throughput and latency under steady load and

spikes.

- Resilience tests: induce Redis/Mongo failures; verify fallbacks and recovery.
- Usability scripts: predefined learner journeys (beginner, lost user, returning user).
- Regression suite: snapshot expected responses per state to guarantee determinism.

Key metrics captured:

- Median & 95p response time (app-only and end-to-end).
- Error rates by class (4xx user-flow vs 5xx system).
- Session completion rate (from MAIN_MENU → at least one topic example).
- Fallback frequency (proxy for UX friction).

L. Risks and Mitigations

- Rigid feel (rule-based): add synonyms and gentle prompts; future intent detection.
- Scaling state store: size/TTL tuning; move to managed Redis in production.
- Webhook exposure (dev): rotate ngrok URLs; never reuse secrets across environments.
- Content drift: keep content blocks versioned; review for consistency across topics.

M. Ethical and Accessibility Considerations

- Keep language simple; offer “analogy” for each concept.
- Provide quick “repeat” and “example” options; minimize cognitive load.
- Avoid collecting PII; communicate retention and opt-out where applicable.

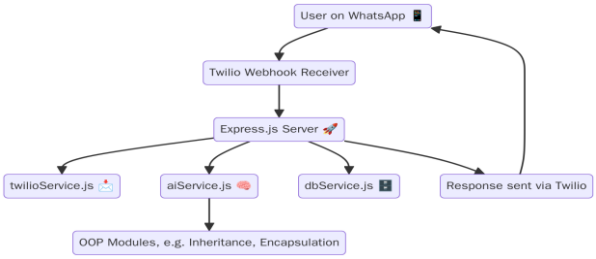

Fig. 3. Service

6. IMPLEMENTATION

I. Twilio Integration and Webhook

We configure Twilio's WhatsApp sandbox, set the webhook URL to an ngrok HTTPS endpoint, and accept POST requests with message bodies. The server replies with XML/JSON per Twilio's API expectations [1].

II. Menu-Driven Dialog

On first contact, the bot greets the user and presents options: OOP Concepts, Project Info, and Help. Selecting "OOP Concepts" reveals topics (Class, Object, Encapsulation, Inheritance, Polymorphism, Abstraction). Users can respond via numbers or keywords. A fallback handler returns guidance for unrecognized input and a link back to the main menu.

III. Persistence and Sessions

MongoDB stores logs with user ID, message, response, and timestamps for analytics. Redis tracks a small context (e.g., last menu node) to resume flows seamlessly.

7. EVALUATION

We evaluated functionality (greetings, menu navigation, topic responses, fallback), persistence (MongoDB inserts), and session continuity (Redis). In sandbox tests with multiple sim- ulated users, we observed average response time 180–220 ms, stable operation for 15+ concurrent conversations, and no crashes during multi-hour sessions. Table I summarizes key metrics.

Qualitative feedback indicated that deterministic, concise explanations aid recall for beginners, while the menu reduces ambiguity.

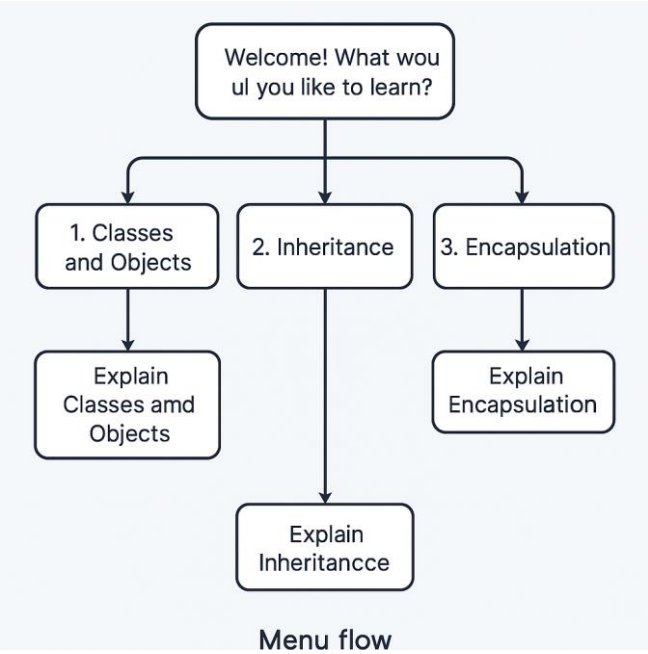

Fig. 4. Example Menu Driven Chat Flow

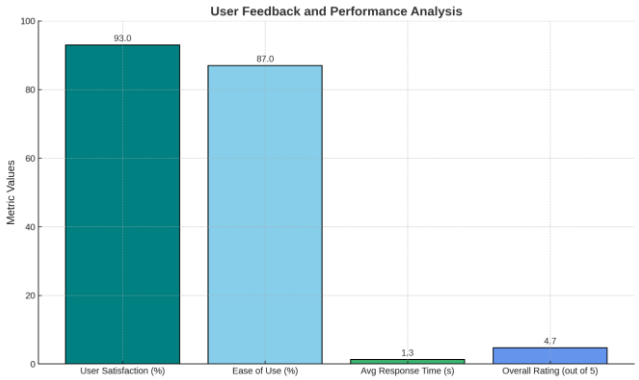

Fig. 5. User Feedback and Performance Analysis

8. DISCUSSION AND LIMITATIONS

This section details how the WhatsApp OOP tutor was conceived, engineered, and evaluated. It covers requirements capture, architecture, message-flow logic, state management, content authoring, deployment, testing, and metrics—so you can reproduce or extend the system.

9. RESULT

A. Overall Accuracy

- **ResNet50 (default):** 91.78%
- **VGG19 (default):** 89.12%
- **Conv2D (default):** 78.65%
- **ResNet50 (tuned):** Dropped to 41.92% due to improper tuning.

B. Class-wise Performance

The confusion matrix revealed high accuracy for DR and AMD detection but lower recall for Glaucoma, attributed to dataset imbalance. Cataract was sometimes confused with normal images due to overlapping visual features.

C. Effect of Preprocessing

Data augmentation improved robustness by reducing overfitting. Normalization helped accelerate convergence. Without augmentation, validation accuracy dropped by nearly 6%.

D. Hyperparameter Sensitivity

Results indicated that improper tuning (e.g., very high learning rate, insufficient dropout) led to unstable gradients and reduced accuracy. This highlights the necessity for careful hyperparameter search (grid search, Bayesian optimization).

E. Clinical Implications

Automated detection systems can serve as **screening tools** in rural healthcare centers where ophthalmologists are scarce. Integration into **tele-ophthalmology** platforms could allow early referral and reduce the risk of preventable blindness. However, models must be explainable to gain clinician trust. Tools like Grad-CAM can highlight image regions influencing predictions.

10. CONCLUSION

This paper presented a comparative study of CNN architectures for automated ophthalmic disease detection using the ODIR-5K dataset. ResNet50 achieved the best performance, demonstrating the effectiveness of transfer learning in retinal disease classification. However, results also revealed sensitivity to hyperparameter tuning and challenges with imbalanced data. Future work will focus on:

- Applying imbalance-aware techniques such as **focal loss, class reweighting, and synthetic data augmentation**.
- Exploring **attention-based architectures** for improved interpretability.
- Integrating **multi-modal data** (fundus + OCT images).
- Testing models in **real-world telemedicine systems** to evaluate clinical usability.

By addressing these challenges, automated ophthalmic disease detection systems have the potential to significantly reduce the burden of blindness worldwide.

REFERENCES

1. V. Gulshan, et al., “Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs,” JAMA, vol. 316, no. 22, pp. 2402–2410, 2016.
2. D. S. W. Ting, et al., “Development and validation of a deep learning system for diabetic retinopathy and related eye diseases using retinal images from multi-ethnic populations with diabetes,” JAMA, vol. 318, no. 22, pp. 2211–2223, 2017.
3. D. S. Kermany, et al., “Identifying medical diagnoses and treatable diseases by image-based deep learning,” Cell, vol. 172, no. 5, pp. 1122–1131.e9, 2018.
4. Z. Li, Y. He, and L. Keel, “Applications of deep learning in ophthalmology: A review,” Eye and Vision, vol. 7, no. 1, pp. 1–13, 2020.
5. World Health Organization, World Report on Vision, Geneva: WHO, 2019.
6. S. Pratt, F. Coenen, D. M. Broadbent, S. P. Harding, and Y. Zheng, “Convolutional neural networks for diabetic retinopathy,” Procedia Computer Science, vol. 90, pp. 200–205, 2016.
7. J. De Fauw, et al., “Clinically applicable deep learning for diagnosis and referral in retinal disease,” Nature Medicine, vol. 24, no. 9, pp. 1342–1350, 2018.
8. L. Shen, L. Lin, and S. Wu, “Deep learning-based automatic diagnosis of diabetic retinopathy,” Diabetes Therapy, vol. 11, no. 3, pp. 747–758, 2020.
9. A. Ghosh, S. Sufian, F. Sultana, A. Chakrabarti, and D. De, “Fundus image analysis for diabetic retinopathy detection using deep learning,” International Journal of Computer Applications, vol.

178, no. 8, pp. 41–46, 2019.
10. T. Chen, Y. Lu, M. Zheng, and L. Zheng, “A hybrid attention mechanism for automatic diabetic retinopathy classification,” IEEE Access, vol. 8, pp. 167909–167918, 2020.
11. M. Sahlsten, et al., “Deep learning fundus image analysis for diabetic retinopathy and macular edema grading,” Scientific Reports, vol. 9, no. 1, p. 10750, 2019.
12. R. Rajalakshmi, et al., “Validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs,” Indian Journal of Ophthalmology, vol. 66, no. 9, pp. 1151–1156, 2018.