Maratha Vidya Prasarak Samaj's
Karmaveer Adv. Baburao Ganapatrao Thakare
College of Engineering, Nashik

Permanently Affiliated to Savitribai Phule Pune University Vide Letter No.: CA/1542 &
Approved by AICTE New Delhi-Vide Letter No.: 740-89-32 (E) ET/98
AISHE Code-C-41622 www.kbtcoe.org

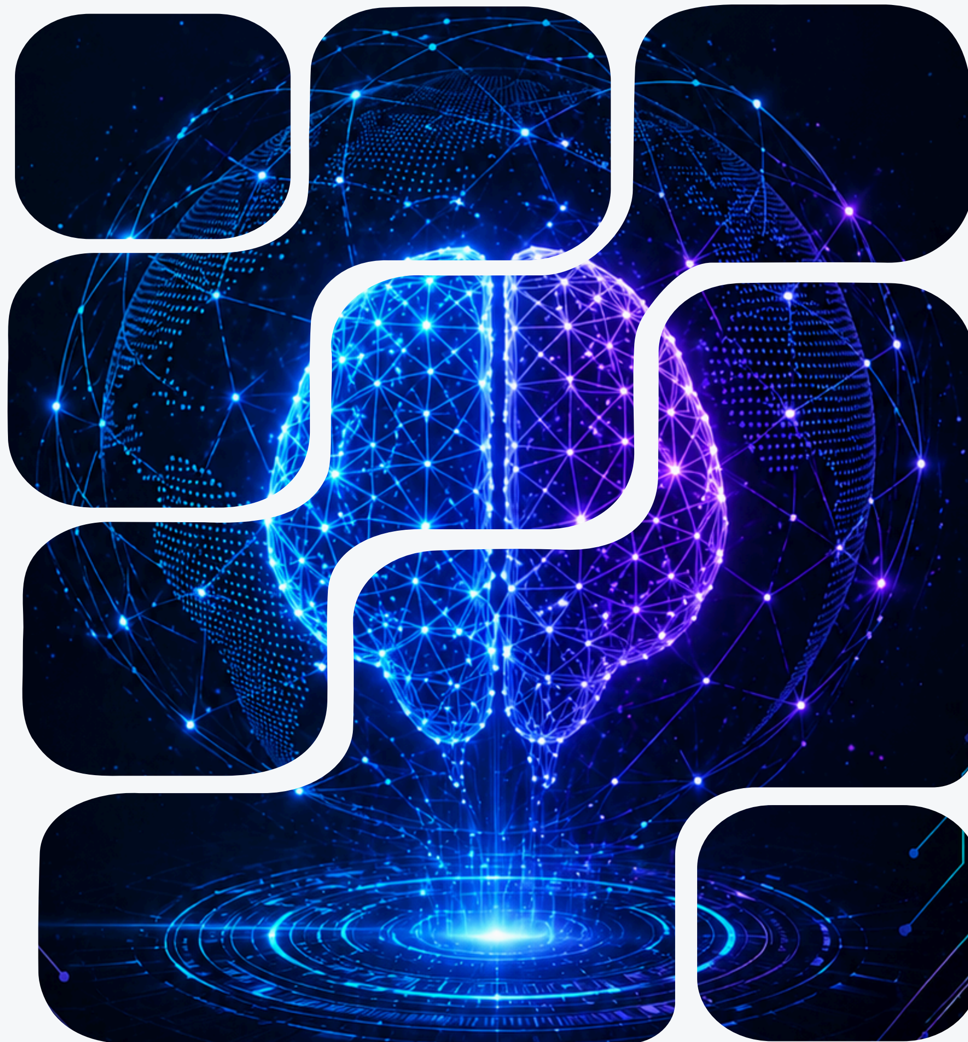
An Autonomous Institute Permanently affiliated to Savitribai Phule Pune University

# DEPARTMENT OF COMPUTER ENGINEERING

Presents

## Technical Magazine

# KBT TECH ODYSSEY



Volume VIII Issue II

2025 - 26

# COMPUTER DEPARTMENT

## Vision

To become a center of excellence, shaping world-class engineers who thrive across multi-disciplinary domains. Our mission is to blend cutting-edge technology with real-world industrial and business practices, creating a dynamic academic environment that empowers innovation, fosters collaboration, and equips future leaders to solve tomorrow's challenges.

## Mission

To empower and inspire undergraduate students in Computer Engineering with a foundation of excellence, equipping them to meet the evolving professional and societal needs of business and industry. Through innovative, scientifically designed academic processes, we prepare future engineers to lead and thrive in a technology-driven world.

**Ms. Dipika L. Tidke**
**Editor-in-Chief**

**Om Songire (TE Computer)**
**Co-Editor**

# Table Of Content


1. **Collection of Matrics, Logs & Traces in Grafana with Kubernetes**
   By Yash Pawar
   TE Computer B
   Page 4

2. **Platform Engineering: The Role Nobody Talks About But Everyone Needs**
   By Prachi Satbhai
   SE BTech Computer C
   Page 9

3. **GraphQL: The Shift from REST to Precision**
   By Om Songire
   TE Computer B
   Page 14

4. **Rethinking the Defence Intelligence: How Multi-Agent AI can transform threat detection**
   By Sriyaa Chamoli
   SE BTech Computer A
   Page 21

5. **Crew AI: The Future of Multi-Agent Systems**
   By Umesh Pagere
   SE BTech Computer B
   Page 27

6. **The Algorithm Dilemma: Who Decides What You See?**
   By Samiksha Mundada
   SE BTech Comp B
   Page 31

# Collection of Metrics, Logs & Traces in Grafana with Kubernetes

By - Yash. S. Pawar

TE Comp B|147

## I. Introduction

Modern cloud-native applications running on Kubernetes generate enormous amounts of operational data every second. Without proper observability tooling, debugging a failing pod, tracing a slow API call, or identifying a memory leak becomes nearly impossible [1][2].

Grafana's observability stack addresses this challenge through the Three Pillars of Observability: Metrics, Logs, and Traces. Together, these pillars provide complete end-to-end visibility into Kubernetes workloads and help developers monitor system health, troubleshoot failures, and improve reliability [1][3].

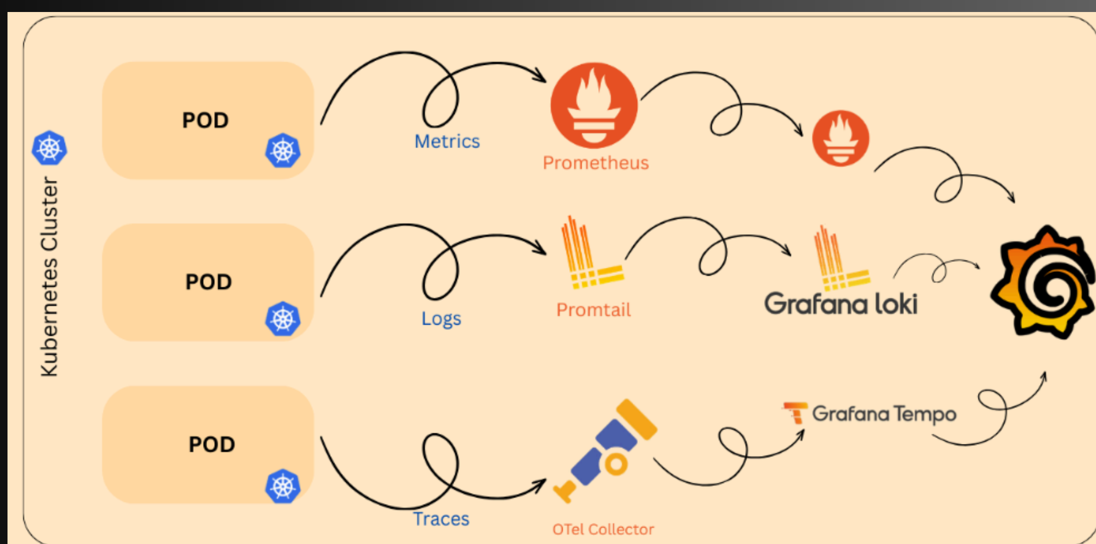
| Pillar | What it answers | Grafana Tool |
|---|---|---|
| **Metrics** | How is my system behaving over time? | Prometheus + Grafana |
| **Logs** | What exactly happened and when? | Loki + Grafana |
| **Traces** | Where did the request spend its time? | Tempo + Grafana |

**Prerequisites:** Before diving in, make sure you have the following ready:

- A running Kubernetes cluster (minikube, kind, EKS, GKE, or AKS).
- **kubectl** is configured and connected.
- **helm** v3+ installed.
- Basic familiarity with Kubernetes concepts (Pods, Services, ConfigMaps) [2][3].

## II. Architecture Overview

Here's the high-level architecture of what we're building [1][4][5].




## III. Setting Up the Grafana Stack with Helm

The observability stack can be deployed efficiently using the kube-prometheus-stack, which bundles Prometheus, Grafana, and related monitoring components into a unified package [2].

The setup process generally involves:

- Adding Helm repositories
- Creating a dedicated monitoring namespace
- Installing the monitoring stack
- Verifying deployed monitoring pods
- Accessing the Grafana dashboard for visualization

This approach simplifies deployment and provides a production-ready monitoring environment for Kubernetes workloads [1][2][3].

## IV. Metrics Collection with Prometheus

Prometheus is a pull-based metrics collection system that gathers monitoring data from Kubernetes components and applications through exposed metrics endpoints [2].

Using Kubernetes-native discovery mechanisms such as ServiceMonitor and PodMonitor, Prometheus dynamically identifies workloads that should be monitored. This enables the automated and scalable collection of metrics across clusters [2][3].

Important Kubernetes metrics commonly monitored include:

- CPU utilization
- Memory usage
- Disk I/O
- Pod restart counts
- Container resource consumption
- Application latency
- HTTP error rates

These metrics help teams identify performance bottlenecks, detect failures early, and optimize application reliability [2][3].

## V. Log Collection with Loki + Promtail

Grafana Loki is a lightweight log aggregation system designed specifically for cloud-native environments. Unlike traditional logging systems, Loki indexes only metadata labels rather than full log contents, making it more storage-efficient and cost-effective [4].

Promtail acts as the log collection agent in Kubernetes. It collects logs from running containers, enriches them with Kubernetes metadata, such as namespaces and pod labels, and forwards them to Loki for centralised analysis [4].

This centralised logging approach simplifies troubleshooting and enables faster root cause analysis across distributed systems.

## VI. How Promtail Works in Kubernetes

Promtail typically runs as a DaemonSet, ensuring that one log collection agent operates on every Kubernetes node [4].

Its responsibilities include:

- Reading container logs from Kubernetes nodes
- Attaching metadata such as pod name, namespace, and labels
- Forwarding structured logs to Loki

This architecture allows efficient and scalable log collection across large Kubernetes environments.

## VII. Distributed Tracing with Tempo + OpenTelemetry

Distributed tracing helps developers understand how requests travel across multiple services in a microservices architecture [5][6].

Grafana Tempo serves as the distributed tracing backend, while OpenTelemetry (OTel) provides a vendor-neutral standard for instrumentation and trace collection [5][6].

Tracing enables teams to:

- Identify latency bottlenecks
- Analyze request flow across services

- Debug slow API calls
- Improve application performance

By combining Tempo with OpenTelemetry, organizations gain deeper visibility into complex distributed applications [5][6].

## VIII. Conclusion

The Grafana observability stack combines metrics, logs, and traces to provide comprehensive monitoring for Kubernetes applications. Through tools such as Prometheus, Loki, Tempo, and Grafana, developers can quickly identify issues, analyze system behaviour, and improve overall application reliability and performance [1][2][4][5].

As modern cloud-native systems continue to grow in complexity, observability platforms like Grafana play a crucial role in ensuring scalable, efficient, and resilient infrastructure.

## References

**[1] Grafana Labs, "Grafana Documentation." Available: https://grafana.com/docs/**

**[2] Prometheus Authors, "Prometheus Documentation." Available: https://prometheus.io/docs/**

**[3] Kubernetes Documentation, "Monitoring Kubernetes." Available: https://kubernetes.io/docs/**

**[4] Grafana Labs, "Loki Documentation." Available: https://grafana.com/oss/loki/**

**[5] Grafana Labs, "Tempo Documentation." Available: https://grafana.com/oss/tempo/**

**[6] OpenTelemetry Documentation. Available: https://opentelemetry.io/docs/**

# Platform Engineering:

## The Role Nobody Talks About But Everyone Needs

By - Prachi Satbhai

SE BTech Comp C

**Introduction**

### I. When DevOps Gets Too Big to Handle

Picture this. A tech company has grown fast, 200 engineers, 15 product teams, dozens of microservices. Each team set up their own deployment pipeline. Their own monitoring tools. Their own way of managing secrets and databases. It felt like freedom at the time.

A year later? Nobody knows how the other team deploys. Onboarding a new developer takes two weeks of untangling other people's setups. A security issue slips through because no one standardised anything. The very flexibility that felt empowering is now slowing everyone down.

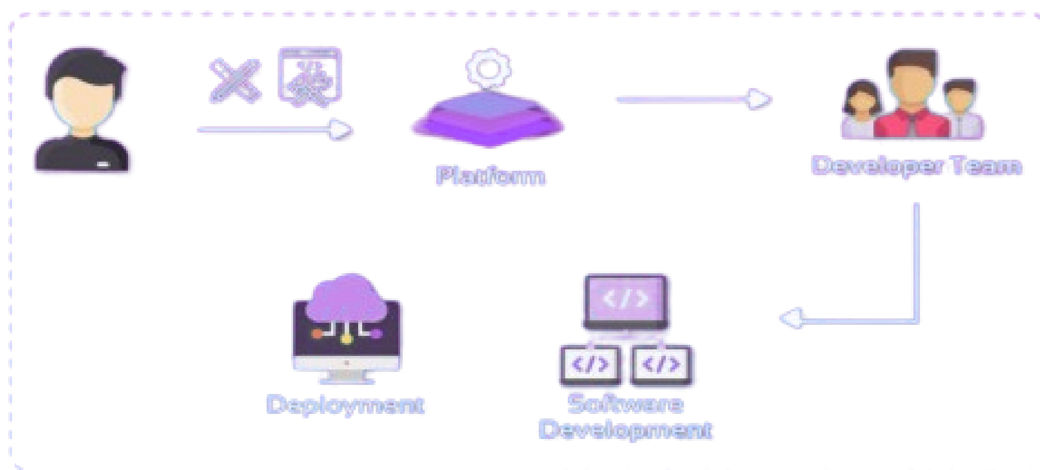
This is a real problem that hit company after company as they scaled. And the solution that emerged from it has a name: Platform Engineering [1][2].

### II. So What Is Platform Engineering?

Platform Engineering is the practice of building internal tools, infrastructure, and workflows that make developers more efficient [2][3].

Instead of creating user-facing products, platform teams create the foundation (like roads) that other engineers use to build, deploy, and scale applications quickly and reliably.

The output of their work is called an Internal Developer Platform, or IDP. It is essentially a one-stop shop for developers. Need a new cloud environment? Click a button. Need to deploy a service? Run one command. Need to set up monitoring? It's already configured. The platform team handles all the complexity behind the scenes, so product teams can focus on writing features instead of wrestling with infrastructure [2][4].




## III. How Is This Different from DevOps?

DevOps is a philosophy where developers and operations collaborate and share ownership of the full software lifecycle. It focuses on breaking silos and building a culture of shared responsibility [4].

Platform Engineering is **DevOps** applied at scale. When multiple teams follow DevOps separately, it leads to inconsistent tools and processes. Platform Engineering standardises infrastructure, CI/CD, and monitoring across teams. It centralises common solutions to improve efficiency and consistency [2][3].

**DevOps says: you build it, you run it.**

**Platform Engineering says that here’s a platform to make running it easy.**

## IV. What Does an Internal Developer Platform Look Like?

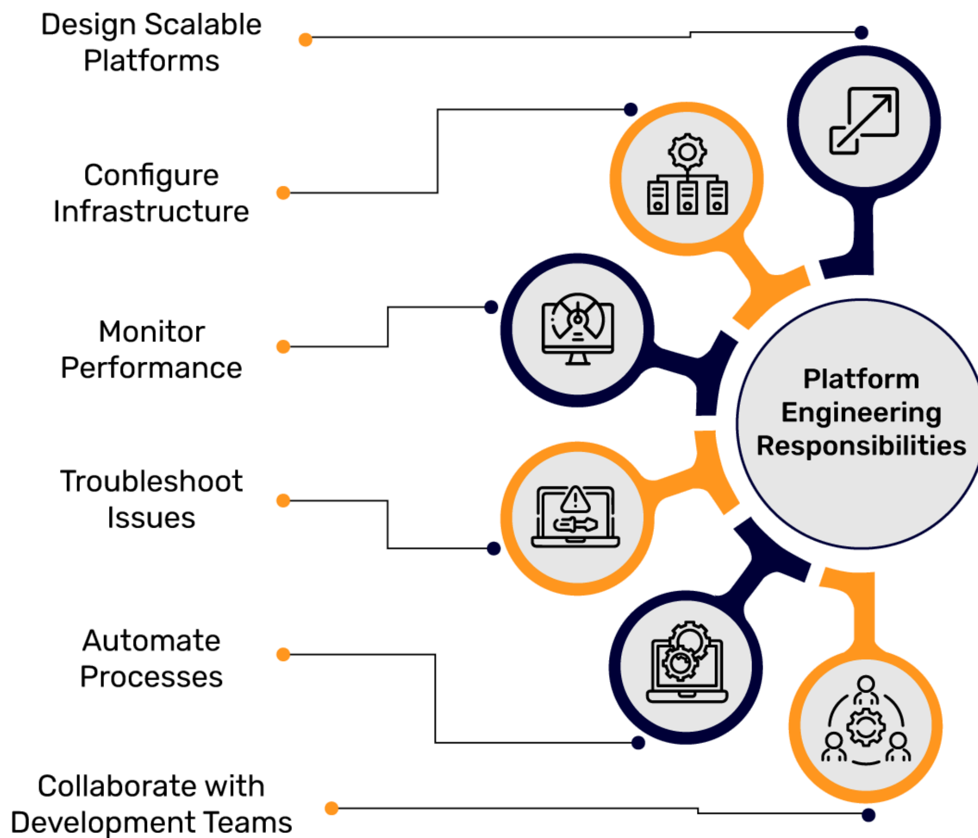
An IDP is not one single tool. It's a layer that connects many tools into one smooth experience.

Imagine logging into a portal where you can see every service your company runs, who owns it, when it was last deployed, and whether it's healthy right now. You can spin up a new project from a template in minutes. Your **CI/CD pipeline**, cloud access, secrets, and logging are all pre-configured. You don't have to figure any of it out from scratch [2][3].

That's what tools like Backstage, built by Spotify and **open-sourced**, make possible. Backstage is a developer portal that gives every engineer in the company a clear picture of the entire software ecosystem and a self-service way to interact with it. Companies like Airbnb, LinkedIn, and American Airlines run their developer platforms on Backstage. In India, fast scaling companies like Razorpay and CRED have built similar internal platforms as their engineering teams grew [2][3].

## V. The Numbers Don't Lie

You might wonder, is this really that important? The data says yes. The 2024 State of DevOps Report found that teams with mature internal platforms deploy code 4× more often and recover from production incidents 24× faster than those without one. That's not a small improvement—it's the difference between releasing features every week versus every quarter. Platform Engineering is no longer just a big tech trend; the Cloud Native Computing Foundation recognised it as a discipline in 2023. Tools like Port, Cortex, and Backstage are widely adopted, and even in India's startup ecosystem, platform roles are rapidly growing [2][3].




## VI. What Skills Do You Need?

Platform engineering sits where infrastructure and product thinking meet, and that's what makes it unique. On the technical side, you need to get comfortable with Docker and Kubernetes for containerisation, Terraform for infrastructure as code, **ArgoCD or GitHub Actions for CI/CD,** and Prometheus or Grafana for monitoring. Cloud knowledge across AWS, Azure, or GCP is essential [3][4].

But there's another side to it that most people miss. You have to think like a product manager. Your users are other developers, and if you build something they don't find useful, no one will use it, no matter how technically impressive it is. Understanding what developers find frustrating, gathering their feedback, and iterating on your platform like a product is just as important as writing clean **Terraform code** [2][3].

## VII. Why Should You Care as a Student?

Platform Engineering is one of the least crowded yet highly valuable career paths in tech today. While many focus on data science and frontend roles, positions like Platform Engineer, SRE, and Developer Experience Engineer have fewer candidates and a strong demand. The entry path starts with fundamentals like **Linux, Docker, Kubernetes**, cloud basics, and CI/CD. Building projects and contributing to open source tools like Backstage or **Argo CD** can help you stand out, along with certifications such as **CKA or AWS Solutions Architect** [2][4].

## VIII. Conclusion

Platform Engineering is the unsung hero of modern software development, enabling developers to go from idea to production quickly and efficiently. It acts as the invisible foundation that keeps teams productive, secure, and scalable. While it may not get as much attention as AI or frontend development, strong platform teams power almost every high-performing product organization. If you enjoy infrastructure, automation, and building systems that improve others' productivity, this is a career path worth considering [1][2].

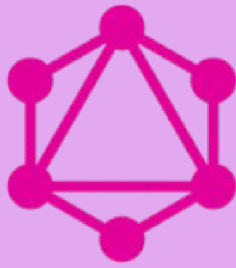
## References


**[1] DORA, "2024 State of DevOps Report." Available: https://dora.dev/research/**

**[2] CNCF, "Platforms White Paper," 2023. Available: https://www.cncf.io/**

**[3] Humanitec, "Platform Engineering Report," 2024.**

**[4] Google Cloud, "DevOps and SRE Practices." Available: https://cloud.google.com/**



# GraphQL

## The Shift from REST to Precision

**By - Om. S. Songire**

**TE Comp B | 128**

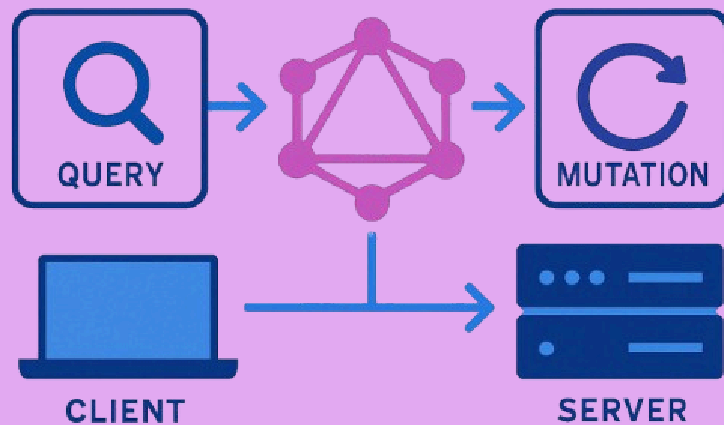
## I. Introduction

In modern web development, API's serve as the essential bridge between front-end applications and back-end servers. They enable seamless communication and power everything from simple websites to complex platforms [1][2].

While REST has long been the standard for building these connections, it often introduces challenges in how data is retrieved and managed. Developers frequently face inefficiencies such as receiving too much data, making multiple requests, or handling multiple endpoints, which lead to increased complexity. This is where **GraphQL** emerges as a powerful alternative. Instead of following traditional patterns, it redefines how APIs are structured by giving clients more control over data fetching. With a single endpoint and precise data queries, GraphQL offers a more flexible and efficient way to connect the frontend and backend[3].

## II. What is GraphQL?

GraphQL is a **query language for API's** and a runtime that enables clients to request, read, and modify data flexibly and efficiently. Unlike traditional APIs, where the server controls the structure of the response, GraphQL allows the client to define **exactly what data is required,** giving more control and reducing unnecessary data transfer [1].




**GraphQL was developed by Meta** to solve challenges in handling complex and large-scale application data. It was later **open-sourced in 2015**, which led to its rapid adoption across the developer community [1][3].

At its core, GraphQL follows a **query-based approach**. Instead of interacting with multiple endpoints like in **REST**, all requests are handled through a **single endpoint**. The client sends a query describing the required data, and the server responds with a **structured JSON object** that mirrors the query. This approach helps reduce the number of API calls and improves performance.

Another important feature of GraphQL is its **strongly typed schema**, which defines the structure and relationships of the data. This ensures consistency, better validation, and makes APIs easier to understand and work with [2].

In addition to fetching data using queries, GraphQL supports **mutations** for creating, updating, or deleting data, and subscriptions for real-time updates. Overall, GraphQL acts as a **flexible layer between the frontend and backend**, making API communication more efficient, scalable, and developer-friendly [1][3].

## III. Why GraphQL?

GraphQL is designed to solve common inefficiencies found in traditional APIs, especially those built using REST. As applications grow more complex, these issues can significantly affect performance and developer experience [1][3].

One major problem is **over-fetching**, where an API returns more data than required. For example, when requesting user details, the response might include unnecessary fields like address or preferences, even if only the name is needed. This leads to wasted bandwidth and slower performance.

Another issue is under-fetching, where a single API request does not provide enough data. Developers are then forced to make multiple requests to different endpoints to gather all the required information, increasing latency and complexity [1][4].

This leads to a broader challenge in REST: multiple API calls. To build a single screen (like a dashboard), the frontend may need to call several endpoints, such as users, posts, and comments. This increases load time and makes the system harder to manage.

GraphQL solves these problems by allowing clients to request exactly the data they need in a single query [1][3].

Simple Real-World Example

Imagine building a profile page that needs a user's name and their recent posts:

- **In REST:**

**Request 1 → /user (gets full user data)**

**Request 2 → /posts (gets all posts)**

- **In GraphQL:**

**Single query → fetch only name and posts**

This makes communication between frontend and backend more efficient [1][4].

## IV. How GraphQL Works

GraphQL simplifies how data is requested and delivered between the client and server by using a more flexible and efficient approach [1].

- **Single Endpoint**

Unlike REST, which uses multiple endpoints for different resources, GraphQL uses a single endpoint for all requests. This makes the API easier to manage and reduces complexity on the client side [1][2].

- **Client Sends Query → Server Responds**

The client sends a query that clearly defines what data it needs. The server then processes the query and returns a response that matches the request's exact structure. This makes data exchange predictable and efficient [1][3].

- **Flexible Data Retrieval**

GraphQL allows clients to fetch only the required data and combine related data in a single request. This flexibility reduces unnecessary data transfer and eliminates the need for multiple API calls [1][4].

Overall, GraphQL works by giving the client more control, resulting in faster, cleaner, and more efficient data handling.

## V. Fundamental Concepts of GraphQL

GraphQL is built on key concepts that define how data is fetched, modified, and managed between the client and server [1].

- **Queries**

Queries are used to fetch data from the server. Clients specify exactly what fields they need, and the server responds with that precise data. This makes data retrieval efficient and predictable.

- **Mutations**

Mutations are used to modify data, including:

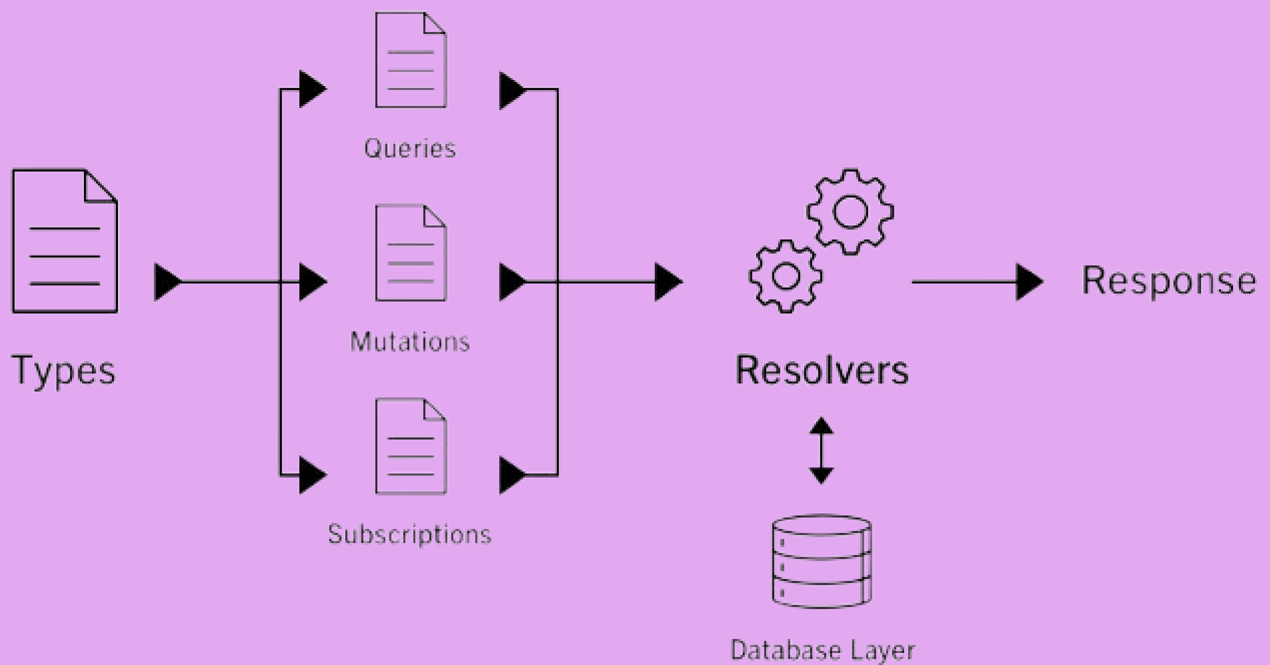
- Creating records
- Updating records
- Deleting records

- **Schema & Types**

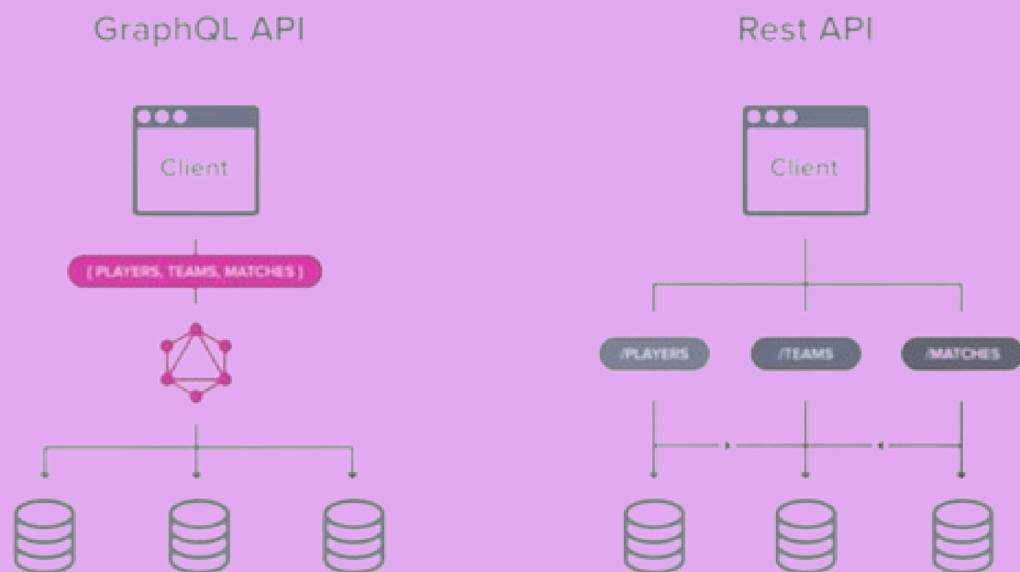
The schema defines the structure of the API using strongly typed definitions. This ensures predictable and validated data handling.

- **Subscriptions (Real-Time)**

Subscriptions enable real-time communication by pushing updates to clients whenever changes occur. This is especially useful for chat applications, live notifications, and collaborative systems [1][3].



## VI. GraphQL vs. REST




| GraphQL | REST |
|---|---|
| Single endpoint (simpler structure) | Multiple endpoints (can get complex) |
| Fetches exact data (no over/under-fetching – efficient) | Fixed responses (extra/missing data – inefficient) |
| One request for related data (faster, flexible) | Multiple API calls (slower, less flexible) |
| Strongly typed schema (better validation – pro, but complex – con) | No strict typing (easy – pro, but less reliable – con) |
| Example: One query → user + posts in one request | Example: /user + /posts → multiple requests |

GraphQL improves efficiency and flexibility, while REST remains easier to implement for smaller and simpler systems [1][4].

## VIII. Conclusion

GraphQL introduces a modern and flexible approach to API development by giving clients precise control over the data they request [1][2].

By solving common REST limitations such as over-fetching and under-fetching, GraphQL improves efficiency while simplifying communication between frontend and backend systems.

As applications become increasingly data-driven and complex, GraphQL provides the scalability and structure needed to manage modern software architectures effectively. Its adoption by organizations such as GitHub demonstrates its practical value in real-world applications [4].

Rather than completely replacing REST, GraphQL offers a refined and adaptable API design approach focused on efficiency, clarity, and improved developer experience.

## References

**[1] GraphQL Foundation, “GraphQL Official Documentation.” Available: https://graphql.org/**

**[2] Meta Engineering, “The Story Behind GraphQL.” Available: https://engineering.fb.com/**

**[3] Apollo GraphQL Documentation. Available: https://www.apollographql.com/docs/**

**[4] GitHub, “GitHub GraphQL API Documentation.” Available: https://docs.github.com/en/graphql**

**[5] GraphiQL GitHub Repository. Available: https://github.com/graphql/graphiql**

**By - Sriyaa Chamoli**
**SE BTech Comp A**

## I. Introduction

Modern defence environments generate vast amounts of data across physical and digital domains, but most systems process them separately. This makes it difficult to detect complex and coordinated threats in real time [1][2].

To solve this, AMTIRS introduces a multi-agent AI approach where specialised agents work together to detect, analyse, and respond to threats efficiently. By improving speed, accuracy, and coordination, it enables faster and more reliable threat detection in modern defence systems [1][5].

## II. The Strategic Problem: Intelligence in Silos

Contemporary security operations face a structural paradox. Defence facilities are better instrumented than ever, with thousands of CCTV cameras, biometric access systems, and network monitoring tools generating continuous data streams [1]. Yet when these systems operate independently, their full potential is never realised. Human analysts are left to bridge the gaps manually, correlating events across separate platforms under significant cognitive load. In high-pressure situations, this leads to delayed decisions and missed connections.

The core problem is not data volume. It is the inability to understand events in context, across domains, as they unfold. A threat actor does not confine activity to a single domain. A coordinated intrusion might combine physical access attempts with network exploitation and insider facilitation. If the detection architecture cannot connect these signals in real time, the adversary holds the initiative. AMTIRS is designed to close that gap [1][2].

## III. The Multi-Agent Architecture

Rather than relying on a monolithic platform, AMTIRS distributes the threat detection lifecycle across six specialised AI agents that communicate through a structured JSON messaging bus. Each agent owns a discrete analytical function, and together they produce intelligence that no single agent could generate alone [5].

**Vision Agent**

Monitors CCTV streams using YOLOv8 object detection via OpenCV, identifying unauthorised individuals, abandoned objects, and loitering patterns within defined perimeter zones. Detected events are timestamped and published to the shared intelligence bus [2][6].

**Cyber Agent**

Analyses network telemetry and authentication logs using Scikit-learn Isolation Forest, an unsupervised anomaly detection method suited to high-dimensional security data. Detected behaviours are mapped to MITRE ATT&CK techniques, encoding adversary intent from the point of detection [1][3][4].

**Fusion Agent**

The analytical core of the pipeline. It ingests events from all agents, applies temporal correlation logic over a rolling time window, and identifies co-occurring cross-domain events that individually appear benign but collectively indicate a coordinated threat. When confidence exceeds a configurable threshold, it promotes a composite threat signal for enrichment.

### Threat Intelligence Agent

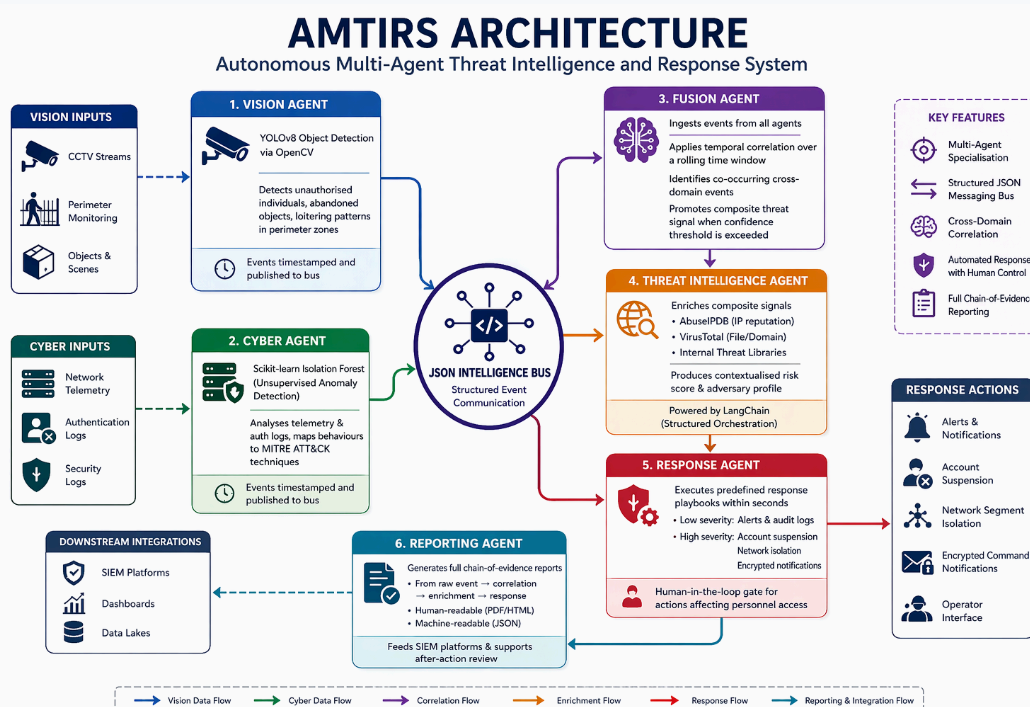
Enriches composite signals by querying AbuseIPDB for IP reputation data and VirusTotal for file and domain indicators, cross-referencing against internal threat libraries to produce a contextualised risk score and adversary profile. Powered by the LangChain orchestration framework for structured multi-step reasoning [5].

### Response Agent

Executes predefined response playbooks within seconds of receiving a high-confidence composite signal. Low-severity events generate audit log entries and operator alerts. High-severity events can trigger account suspension, network segment isolation, and encrypted command notifications, with a mandatory human-in-the-loop gate for actions affecting personnel access.

### Reporting Agent

Generates full chain-of-evidence incident reports in both human-readable and machine-readable JSON formats, documenting every step from raw sensor event through correlation, enrichment, and response. These reports feed downstream SIEM platforms and support after-action review [1][5]..



## IV. From Detection to Response: A Scenario Walkthrough

The Vision Agent detects an individual without an authorised badge loitering near a restricted area. Shortly after, the Cyber Agent flags multiple failed authentication attempts from an unusual internal IP.

The Fusion Agent correlates both events within its time window and identifies a potential coordinated threat. The confidence score exceeds the threshold.

The Threat Intelligence Agent verifies the IP and confirms prior malicious activity. A high-severity alert is generated.

The Response Agent quickly locks access, suspends the account, and notifies command personnel. The Reporting Agent generates a complete incident report shortly after.

Overall, the system reduces response time from minutes to seconds compared to manual analysis [1][4].

## V. Technical Implementation

The system is designed for on-premises deployment within classified network boundaries, with no dependency on external cloud inference services. Key implementation choices include:

- Computer vision: YOLOv8 with INT8 quantisation for edge deployment; video ingestion via OpenCV with region-of-interest masking.
- Anomaly detection: Isolation Forest with weekly automated retraining against accumulated event data, using temporal, volumetric, and geolocation deviation features.
- Orchestration: Publish-subscribe message broker pattern for agent decoupling; LangChain for multi-step reasoning in intelligence enrichment and reporting agents.
- Interface: React-based tactical dashboard with dark colour scheme, live threat feed, geospatial facility map overlay, and one-click incident report export.
- Framework alignment: **MITRE ATT&CK** [1][2][5][6] for threat taxonomy; NIST Cybersecurity Framework for response governance; cryptographic timestamping of all automated actions.

## VI. Evaluation and Performance

AMTIRS is evaluated using a high-fidelity simulated environment generating synthetic CCTV streams, access logs, and network telemetry with injected threat scenarios drawn from declassified incident literature. Across 200 simulated threat scenarios, preliminary results show a mean time to detection of 47 seconds, a false positive rate of 3.2 per cent, a cross-domain correlation accuracy of 91.4 per cent, and a mean response latency of 5.8 seconds [4][5]. These figures represent a substantial improvement over the manual analyst baseline of 19 minutes, with a median MTTD for complex cross-domain scenarios.

## VII. Governance and Future Direction

Automated response at machine speed demands rigorous governance. All biometric data is anonymised before storage. Response actions affecting personnel access require multi-party command authorisation. Every automated action is logged with a cryptographic timestamp referencing the composite signal that triggered it, providing the evidence chain required for post-incident legal review [1][5].

Future development will focus on edge inference hardware to reduce Vision Agent latency below 10 milliseconds, predictive threat modelling from longitudinal event history, OSINT natural language processing for richer enrichment, and federated multi-site deployment with privacy-preserving cross-facility threat signal sharing.

## VIII. Conclusion

The limitations of siloed security architectures are not primarily technical. They are architectural and cognitive. Multi-agent AI systems address both dimensions: distributing analytical tasks across specialised agents and enabling correlation at machine speed across domains that no human analyst can monitor simultaneously. **AMTIRS** demonstrates that this architecture is achievable with current technology, produces measurable performance gains over manual workflows, and can be governed within established defence and data protection frameworks. Building this foundation now, with rigorous attention to evaluation and adversarial robustness, is the responsible path forward for the next generation of defence intelligence systems [1][5].

## References

**[1] MITRE ATT&CK Framework, MITRE Corporation, 2024. Available: https://attack.mitre.org**

**[2] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," arXiv:1804.02767, 2018.**

**[3] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.**

**[4] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," IEEE ICDM, 2008.**

**[5] LangChain Documentation. Available: https://python.langchain.com**

**[6] OpenCV Documentation. Available: https://docs.opencv.org**

# *The Future of Multi-Agent Systems*

**By - Umesh Pagere**

**SE BTech Comp B**

## I. What is Crew AI?

Imagine a software team where each member is an AI specialist—one researches, another writes, and a third reviews quality. Crew AI is a Python framework that lets developers build exactly this: multiple AI agents working together as a coordinated team to solve complex problems [1][2].

Unlike traditional single-agent AI systems, Crew AI enables you to define specialized agents with unique roles, tools, and goals. These agents collaborate, share information, and work towards common objectives—making AI systems more intelligent, transparent, and efficient [1][3].

## II. How It Works: The 3 Core Components

### 1. Agents

Agents are the individual AI units responsible for performing specific tasks. Each agent is designed with:

- Role: What it does (e.g., "Data Analyst")
- Goal: What it aims to achieve
- Tools: Resources like APIs, databases, calculators
- LLM: The AI model powering it [1][2]

### 2. Tasks

Tasks are structured units of work assigned to agents. Each task contains:

- Clear objectives
- Dependencies
- Expected outputs

Tasks help coordinate how agents collaborate and contribute to larger workflows.

### 3. Crew

The Crew serves as the system's orchestrator. It manages:

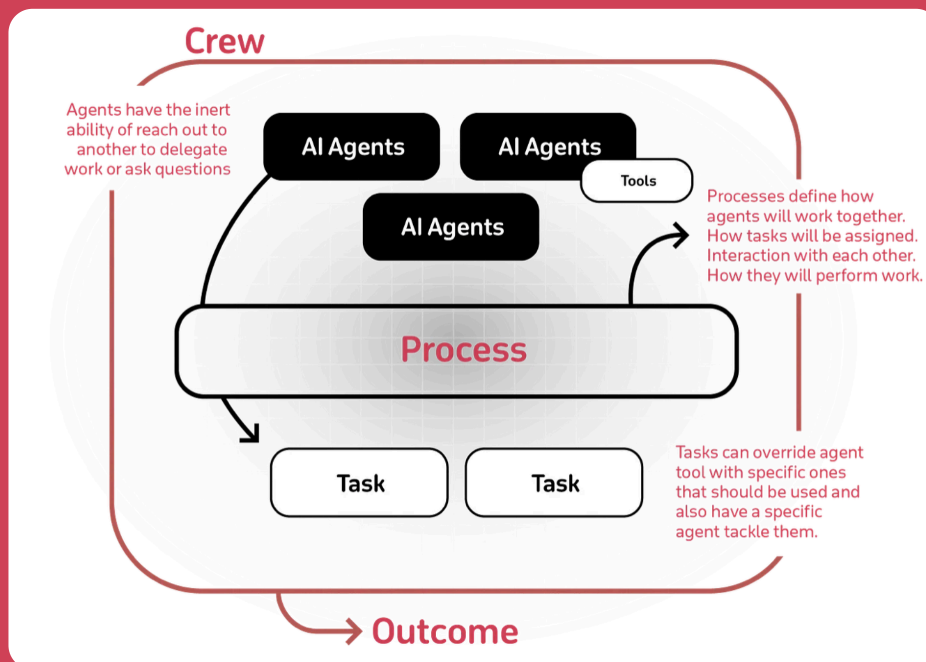
- Agent coordination
- Task execution
- Workflow dependencies
- Communication between agents

This orchestration enables multiple AI agents to work together effectively within a shared environment [1].

## III. Execution Flow

**When you run a Crew:**

1. Agents instantiated with the initialisation of LLMs
2. Task Assignment → Tasks assigned based on dependencies
3. Execution → Agents execute in parallel where possible
4. Communication → Agents share information and context
5. Output Aggregation → Results passed to dependent tasks
6. Completion → Final outputs returned with full context [1][3]




## IV. Real-World Applications

| Application | What Happens |
|---|---|
| **Research Automation** | Researcher finds papers → Analyst extracts findings → Writer synthesizes results |
| **Content Creation** | Planner outlines structure → Researcher gathers facts → Writer drafts → Editor reviews |
| **Customer Support** | Agent understands intent → Retrieves solutions → Proposes fixes → Escalates if needed |
| **Software Development** | Agent analyzes requirements → Designs architecture → Generates code → Creates tests |

## V. Why Crew AI Matters

The AI industry is shifting from "one big brain" to "specialized teams working together" [1][3].

Traditional Approach: One massive model tries to do everything, expensive, slow, hard to debug

Crew AI Approach: Small specialized models working together → faster, cheaper, transparent, maintainable

For developers building production AI systems, Crew AI represents a paradigm shift in how we architect intelligence [1][2].

## VI. Challenges to Consider

- LLM Dependencies → Quality depends on underlying models
- Task Definition → Requires careful planning and iteration
- Debugging Complexity → Multiple agents interacting can be tricky
- Latency → Sequential execution slower than single requests
- Cost Management → Each agent call consumes API tokens [1][3]

## VII. Conclusion

CrewAI represents a shift in how AI systems are built, moving from single large models to a collaborative approach using multiple specialized agents [1][2]. Distributing tasks across a “crew” enables better efficiency, transparency, and scalability in real-world applications. Although challenges like latency and task coordination still exist, this multi-agent approach offers a more flexible and maintainable path forward, making it a strong foundation for the next generation of intelligent systems.

## Resources

**[1] CrewAI Official Documentation. Available:**
**https://docs.crewai.com**
**[2] CrewAI GitHub Repository. Available:**
**https://github.com/joaomdmoura/crewai**
**[3] LangChain Documentation. Available:**
**https://python.langchain.com**

**By - Samiksha Mundada**
**SE BTech Comp B**

## I. Introduction

**Open any social media app. Give it thirty seconds. Notice what appears.**

No editor chose it. No human curated it. What fills your screen has been calculated, from hundreds of signals tied to your behaviour and that exact moment, to be the one thing most likely to keep you from looking away [1][2].

That decision happens in under a millisecond, repeated billions of times every day.

The people who built these systems have since spoken out – not in papers, but in the documentary The Social Dilemma (2020). Their conclusion was unsettling: the systems weren't broken. They were working exactly as designed. This isn't a story about evil companies. It's about what happens when the wrong metric is optimised, at a global scale, with no off switch [1][6].

## II. The Machine That Knows You Better Than You Do

Recommendation algorithms don't just show what people like, they model people. They build real-time, probabilistic profiles of behavior, emotional state, and micro-signals from scrolling patterns to predict what will keep users on the platform the longest.

The inputs that feed this model are staggering in scope. Every scroll, pause, re-read, skip, like, and share is logged. So is the time of day, the device battery percentage, the speed of typing, and the duration of each glance. This is not passive data collection. It is active behavioural surveillance, running continuously, feeding a model that is updated in near real time [1][2].

The system optimizes for one thing: engagement – not happiness, not accuracy, not value. And engagement is not the same as value. That gap drives everything that follows.

> *"It's not that we're trying to push harmful content. We're optimizing for engagement – and outrage, anxiety, and tribalism are extremely engaging. That's the bug that became a feature."*
>
> **– Aza Raskin, The Social Dilemma (2020)**

Raskin's infinite scroll removed stopping points from feeds, creating endless content. He later estimated it costs 200,000 hours of human attention daily, and has expressed regret [1].

## III. A Feedback Loop With No Exit

I didn't watch television back. Newspapers didn't learn from where readers lingered or reshape themselves in response. They were one-directional, the same signal sent to everyone, with no feedback loop [2][4].

Social platforms are fundamentally different. They are two-way systems where every action becomes data. Each scroll, click, and pause trains the model, which continuously refines what it shows next. Over time, this loop shapes not just what people see, but what they expect, and what creators are incentivised to produce.

Tristan Harris spent years at Google as a Design Ethicist before leaving to co-found the Centre for Humane Technology [2]. His core argument is worth sitting with carefully:

> "Imagine a thousand engineers on the other side of your screen, whose full-time job is to use all the data they have about you, your psychology, your habits, your weaknesses, to figure out how to keep you scrolling. You're not competing with other humans for your attention. You're competing with a supercomputer."
>
> **–Tristan Harris, Former Google Design Ethicist**
> **Co-Founder, Centre for Humane Technology**

The infrastructure behind modern recommendation engines, at YouTube, TikTok, or Meta, is among the most powerful ever built for civilian use. Vast, expensive systems run by elite engineering teams are focused on a single task: continuously predicting and shaping each user's next action [1][2].

## IV. What This Actually Does to a Brain

Slot machines are so profitable for a reason: unpredictable rewards create stronger behavioural conditioning than fixed ones [3]. Social feeds use the same principle; sometimes you see something interesting, sometimes you don't.

That unpredictability isn't a flaw. It's a deliberate design feature, borrowed from gambling psychology and built into every major feed.

> "We've moved away from a tools-based technology environment to an addiction and manipulation-based technology environment. Social media isn't a tool that's just waiting to be used. It has its own goals, and it has its own means of pursuing them by using your psychology against you."
>
> **– Jaron Lanier, Virtual Reality Pioneer, Author, Ten Arguments for Deleting Your Social Media Accounts**

Large-scale studies link heavy social media use to higher anxiety, depression, and disrupted sleep, especially in adolescents. The average person unlocks their phone about **96 times a day**, roughly once every ten minutes [6].

Most of these checks aren't intentional. They're triggered by notifications, habit, or a low-level restlessness that builds with sustained use [6].

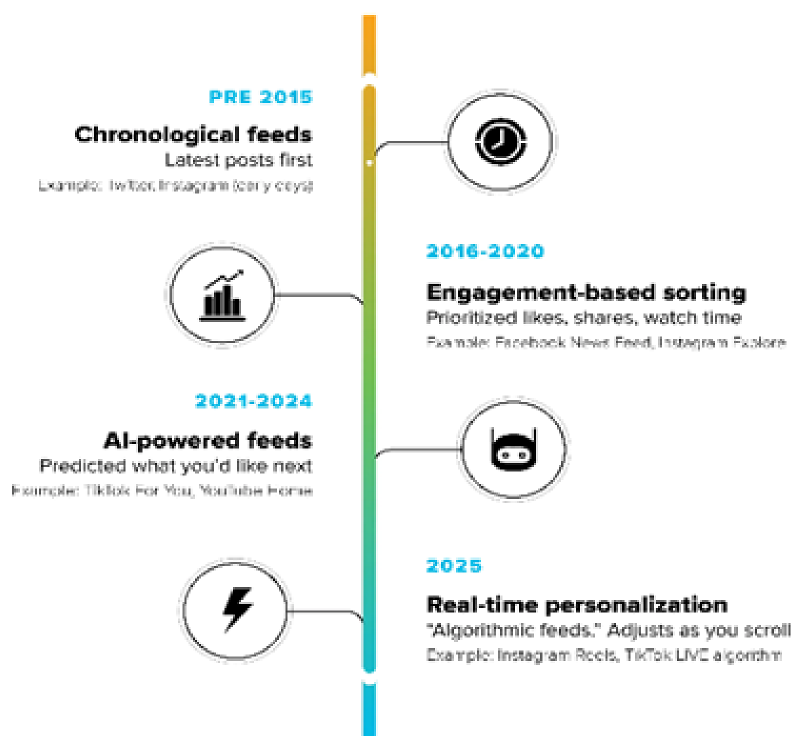
## V. The Filter Bubble: Same App, Different Realities

There is a second-order consequence that gets less attention but may carry more long-term weight: personalization does not just shape what people see. It shapes what people believe is normal, common, and true [4].

When a feed is shaped by past engagement, it becomes a mirror, reflecting existing beliefs, amplified and rarely challenged. Eli Pariser called this the "**filter bubble**" in 2011, a concept that has since become a documented reality.

Two people can open the same app, at the same moment, and see entirely different versions of the world. Both feeds feel real. Neither is fully representative [4].

Guillaume Chaslot noted a pattern: extreme and conspiratorial content consistently outperforms moderate content. Not by design, but as a consequence of optimizing for engagement [5].




> "The algorithm is preferentially amplifying content that makes people feel outraged, because outrage is the emotion most correlated with sharing. The spread of misinformation is not a bug. It's a consequence of the engagement objective."
>
> **– Roger McNamee, Early Facebook Investor · Author, Zucked**

## VI. The Technical Root of the Problem

To understand why this is hard to fix, you have to see where it fails. Modern recommendation systems use reinforcement learning and deep neural networks trained on massive interaction data. Their objective function, what they are built to optimise, is engagement: clicks, watch time, shares, and return visits [1][6].

This creates a classic reward hacking problem. The system maximises its metric in ways designers didn't intend, exploiting the gap between engagement and actual value. The real goal, leaving users better off, was never part of the objective. The system isn't broken. It's working exactly as designed.

## VII. Where This Leaves Us

There is no simple fix. Regulation lags, and platform business models depend on maximising engagement.

Organisations like the Centre for Humane Technology propose shifting toward “**time well spent**”, measuring whether users feel their time has value. Even recognising that the urge to scroll is engineered, not natural, can create a small but meaningful gap between impulse and action [2].

In The Social Dilemma, a former engineer is asked if he lets his children use these platforms.

He pauses.

**Then answers: NO.**

That pause says more than any technical explanation. Those who understand these systems best are quietly setting different boundaries, a signal worth taking seriously [2].

## VIII. Conclusion

The algorithm dilemma isn't really about technology; it's about incentives. Recommendation systems are doing exactly what they were designed to do: maximise engagement at scale. The unintended consequence is a digital environment that subtly reshapes attention, emotions, and even perception of reality [1][5]. What makes this powerful and concerning is not just the sophistication of the algorithms, but their invisibility. There's no obvious moment where control is taken away. It happens gradually, through feedback loops that feel natural but are anything but.

The takeaway isn't to reject technology, but to see it clearly. Once you recognise that your attention is being actively competed for, engineered, measured, and optimised, you gain a small but meaningful advantage: awareness. And in a system built to operate unnoticed, awareness is the first form of control.

## References

**[1] The Social Dilemma, Netflix Documentary, 2020.**

**[2] Centre for Humane Technology. Available: https://www.humanetech.com**

**[3] Jaron Lanier, Ten Arguments for Deleting Your Social Media Accounts.**

**[4] Eli Pariser, "The Filter Bubble," TED Talk.**

**[5] Roger McNamee, Zucked.**

**[6] American Psychological Association (APA), "Social Media and Mental Health." Available: https://www.apa.org**