Ministry of Education and Science of Ukraine

National University of "Kyiv-Mohyla Academy"

Department of Informatics of the Faculty of Informatics

Development and Implementation of a Military Technology Trends Monitoring System

Text part to the thesis in the specialty "Software Engineering" - 121

Thesis Supervisor:

Senior Lecturer

Andrew Kurochkin

_____ (Signature)

"___" _____ 2025

Done by Student:

Oleksandr Prokhorov

"___" _____ 2025

Kyiv, 2025

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"
Department of Informatics of the Faculty of Informatics

Approved
Head of Department of Informatics,
Associate Professor, Ph.D.
S. S. Gorokhovskyi

_____ (Signature)

"___" _____ 2025

INDIVIDUAL TASK for thesis
of the student Oleksandr Prokhorov
4th year of the Faculty of Informatics
topic: Development and Implementation of a Military Technology Trends Monitoring System

The content of the PM to the thesis:
Abstract
Introduction
Related work
Approach
Solution
Evaluation
Further work and Conclusions
Bibliography

Date of issue: "___" _____ 2025

Supervisor: _____ (Signature)

Task received: _____ (Signature)

Abstract

This work presents the design and implementation of a system for monitoring technological trends in the military sector using Telegram as a data source. The system automatically collects, processes, and analyzes both historical and real-time posts from selected Telegram channels, focusing on the emergence and dissemination of key terminology such as *"реб"* (eng.: *"electronic warfare"*) in our evaluation case study. A modular architecture was developed, combining Go-based data scraping, Python-based aggregation and keyword analysis, and a Grafana dashboard for visualization. The system supports both local Docker-based deployment and cloud-based deployment via Terraform on AWS. Evaluation included performance benchmarks, peak resident-set size (RSS) profiling, and a case study comparing our system's findings against professional media and Google Trends. Results indicate that a Telegram-based pipeline can detect rising interest in electronic-warfare topics earlier than traditional information channels.

Table of Contents

Chapter 1

Introduction

1.1 Background

This research is related to monitoring and retrieving information on the trends in military technologies. Internet and public resources become overflooded with all kinds of data and information, which makes it hard to mass-monitor.

By 2025, Telegram[1] channels have become one of the most comprehensive and diverse sources of up-to-date information[2] on military technologies and related trends. These channels[1] combine real-time updates, expert opinions and user-generated content, offering a unique and diverse resource for researchers, analysts, and journalists alike.

For this work we have selected 11 popular Telegram channels. Each channel averages more than 450 posts per month and boasts over 13,000 posts in total. As illustrated in Table 1.1.

The sheer volume of data underscores the need for advanced, automated approaches to information retrieval and analysis. Manual processing of such vast quantities of data is not only time-consuming but also prone to errors.

[1] Hereafter, the term *channel(s)* refers specifically to Telegram channel(s).

Channel tag	Total posts	Posts per month
@mil_inua	37 371	765
@TyskNIP	14 303	410
@serhii_flash	4 603	227
@VictoryDrones	34 608	1, 629
@btvt2019	17 485	713
@vanguard_spacepro	14 114	394
@mag_vodogray	11 196	301
@samopalwar	2 029	20
@russianocontext	5 429	319
@war_home	3 629	321
@firstdivisionofficial	7 036	284
Average	13 800	489

Table 1.1: channels' posting statistics by *01/29/2025*

1.2 Motivation

The large number of different channels provide varied information on similar topics. Each channel incorporates its own unique artistic style or offers additional analytical insights, making it challenging to determine whether different posts address the same subject or contain crucial new details.

By 2025, many organizations and government bodies had established personal dedicated analytics teams tasked with monitoring emerging posts and news[3]. Their goal is to promptly generate reports and respond to critical information leaks in real time.

To automate this processes and facilitate monitoring of popular and frequently discussed topics, we have developed this service. It provides comprehensive metrics regarding channels, including their reach and overall post activity. It is possible to see aggregated data over extended periods of specific time frame, enabling the identification of trending topics and the analysis of general tendencies over time.

1.3 Objectives and Scope

Monitoring systems are widely deployed across various domains, including healthcare, natural sciences, and energy, among others. In this work, we focus on news monitoring, which is sometimes also referred to as media monitoring.

The primary objective of this work is to investigate the feasibility of developing a news monitoring system focussed on miltech related channels, identify its minimum requirements, and determine the most suitable design solutions for its implementation.

During this research, we have developed and deployed a monitoring system that collects data from selected Telegram channels. The system stores, aggregates, and visualizes collected data through an intuitive dashboard powered by Grafana[4].

This paper focuses on identifying the essential components of such a system and exploring which types of data can be extracted from posts without resorting to high-load computations. Additionally, it addresses effective methods for presenting and interacting with this information to facilitate timely decision-making[2].

1.4 Thesis structure

In Chapter 2 we reviewed some of the related works on news monitoring. Then proposed approach was introduced in Chapter 3. Our solution described in Chapter 4. Then it was evaluated in Chapter 5. Further work possibilities and Conclusions were presented in Chapter 6.

[2]Meaning that the system assists human users in making informed decisions by providing valuable information.

Chapter 2

Related work

2.1 Overview of Monitoring Systems

Most monitoring systems fundamentally perform three essential functions: data scraping, aggregation, and presentation. While these systems share a similar core framework, they are tailored to address a variety of challenges and requirements across different domains.

Depending on their intended application, monitoring systems typically exhibit several common capabilities:

- Collecting data from one or multiple sources.
- Transforming, annotating, or performing preliminary analyses on the collected text.
- Identifying relationships between related stories or data points.
- Presenting information through effective visualizations or dashboards.

2.2 Application of Monitoring Systems

One common application type of Monitoring Systems is the summarization of news to eliminate redundant or repeated information[5]. Given the sheer volume of stories and the variety of sources available on the Internet, readers often struggle to determine whether a story introduces a new topic or merely repeats content they have already seen. This challenge has lead to the development of systems that automate the processes of analysis, filtering, and recommendation of fresh news.

Another significant application is the analysis of the mediasphere. Systems developed for this purpose, such as the NOAM platform[6], could be highly valuable in both social and scientific research. These systems typically integrate various NLP, ML or Aggregation technologies to perform tasks including automatic translation, preliminary annotation, clustering of related content, and providing a unified user interface for querying aggregated data.

While the detection of breaking news and short-term trends[3] might be seen as an extension of mediasphere analysis, we treat it as a distinct application because it focuses on identifying emergent information. The Cambridge Dictionary defines *breaking news* as "information that is being received and broadcast about an event that has just happened or just begun"[7]. Such time-sensitive data can become crucial in the process of analysing and extracting valuable information relevant to targeted topics.

2.3 General Design and Architecture Concepts

On the surface monitoring systems perform data collection tasks, so it require a form of storage to manage the acquired information, what could be considered as core of such application. Depending on the structure and requirements of the collected data, this storage can be implemented using relational databases, non-relational databases, knowledge bases, or other storage solutions [6].

As previously discussed, these systems execute a variety of tasks that should operate independently to prevent mutual performance degradation. Consequently, a modular architecture with pipelines is commonly employed [3, 5, 8]. This design ensures that each component can perform its specific function without being affected by potential delays or issues in other modules. Typically, the database serves as the central shared element among these modules, facilitating smooth data transformation, especially in tasks related to Natural Language Processing (NLP) or First Story Detection (FSD).

In summary, data storage and modular design are pivotal features of monitoring systems, enabling efficient and scalable operations.

2.4 Monitoring Military Trends

Monitoring military trends is not a new field; however, until recently it was primarily performed by dedicated personnel or journalists. In the era of rapidly evolving technologies and diverse information sources, traditional methods have become increasingly harder to perform.

Since the onset of the russian full-scale invasion on Ukraine, Telegram channels have emerged as one of the most frequently used sources of information. According to a study by Gradus Research Company, in 2024 Telegram was the top messenger application in Ukraine [9], at the same year it was regarded as one of the most popular sources of news and information, with approximately 73.4% of Ukrainians using it [10]. These findings underline the significant role and widespread use of Telegram as an information source.

For military and miltech news, Telegram offers an efficient platform. Both large media outlets and individual channels, operated by soldiers or civilian observers, closely following military events and contribute valuable content.

Although some channels focus on aggregating information and providing analyses of new technologies and their applications, much of this work is still carried out manually by people.

Therefore, developing a tool that can analyse, monitor, and identify trending topics in real time would be both beneficial and essential.

Chapter 3

Approach

3.1 Telegram API

A crucial component of this project is data scraping from Telegram, which makes its API particularly important. Telegram offers three types of APIs for developers [1]:

1. **Bot API** – enables the development of applications that interact via Telegram messages.
2. **Telegram API and TDLib** – facilitates the creation of custom clients.
3. **Gateway API** – used for sending verification codes.

In this work, we focus only on the second type of API. This decision was made because the Bot API has limitations when it comes to requesting large volumes of data, managing systematic API calls and enrolling to private channels. Although TDLib also has certain constraints in handling requests, it nevertheless offers greater flexibility and broader capabilities for data scraping.

Telegram API and TDLib (Telegram Database Library) provides a collection of tools for developing custom applications that interact with Telegram. Notably, it allows users to log in as regular users, thereby granting developers access to the full range of functionalities available in Telegram's web and mobile versions.

Initially, we implemented a basic login mechanism using a phone number and a verification code sent by Telegram. However, we encountered an issue: each login attempt created a new connection and session, and after approximately five to ten repetitions the account was automatically logged out and prevented from logging in due to suspicious activity. We contacted Telegram support regarding this matter and the account was eventually unbanned after a few days.

To prevent this issue from occurring again, we opted to store a *session* – a JSON object containing the credentials – which allowed us to log in without repeated authorization.

This solution effectively addressed the login problem, enabling us to proceed with client integration.

Our next step was to extract all the channels to which the Telegram account was subscribed. The extraction was successful: the API returned a `MessageDialogsClass` object, from which we extracted `Dialog` instances. These instances were then cast to `Channel` objects, allowing us to obtain valuable information such as the channel title, name, date of creation, and other relevant metadata. Additionally, we retrieved the `AccessHash` and `ID` for each channel, which proved essential for scraping posts.

To scrape posts, Telegram provides the method `messagesGetHistory`. This method enables developers to specify the channel (by providing its `ID` and `AccessHash`), set a limit (the number of messages to extract per request), and fields to specify logic of posts extraction. Although Telegram's documentation is limited to the RPC schema of methods and potential error codes[1], it is recommended not to exceed 1000 messages per request. Moreover, continuous repeated requests will result in the `420:FLOOD_WAIT` error. In our case, a limit of 100 messages per request proved to be the most efficient configuration, as it minimized the occurrence of the mentioned error.

The Scraper defines four engagement metrics—*views*, *forwards*, *comments*, and *reactions* (with each reaction type recorded separately). To retrieve these metrics, the service invokes the `channelsGetMessages` method, supplying the channel's `ID`, `AccessHash` and an array of target message `ID`s. In order to respect Telegram's rate limits, requests are issued in batches of up to 50 message IDs. Upon receiving the response, the client iterates through each message's `reactions` field, categorizing entries as either standard reaction types or custom emojis, and then persists the parsed metrics via the `Store` interface.

3.2 Subsystems Details

3.2.1 Scraper

The *Scraper* is implemented as a standalone, containerized microservice that exposes a RESTful API for initiating Telegram crawling operations. An external scheduler periodically issues HTTP requests to the service endpoints, each spawning a new goroutine; this design ensures high concurrency and fault isolation.

Initialization. Upon startup, the service performs the following steps:

1. Establishes a connection to the PostgreSQL database and applies any pending schema migrations.

2. Registers HTTP routes on the internal router.

[1] See Telegram API Methods (visited on 04/13/2025)

3. Instantiates the `Crawler` client using the credentials `APP_ID` and `APP_HASH`, together with a designated `SESSION_STORAGE` path.

4. Launches the `Crawler` in the background and attempts to restore a previous session; if none exists, it completes a new login via Telegram's authorization code.

A graceful shutdown handler intercepts termination signals, allowing in-flight tasks to complete and ensuring that all crawler routines terminate cleanly before the service exits.

Scraping Workflow. The primary responsibility of the Scraper is to collect channel metadata, message content, and user engagement metrics.

Upon receiving a crawl request:

1. The handler invokes the `Store` interface to retrieve a list of target channels and their latest processed message identifiers.

2. If a last-seen identifier exists, the `Crawler` fetches all messages posted since that identifier; otherwise, it fetches messages from a predefined historical start date up to the current time.

3. Retrieved messages and associated engagement data are persisted via the `Store`, which wraps all database operations in safe, idempotent transactions.

4. Finally, the endpoint responds with `HTTP/1.1 200 OK` to confirm successful completion.

The overall sequence is illustrated in Figure 3.1.

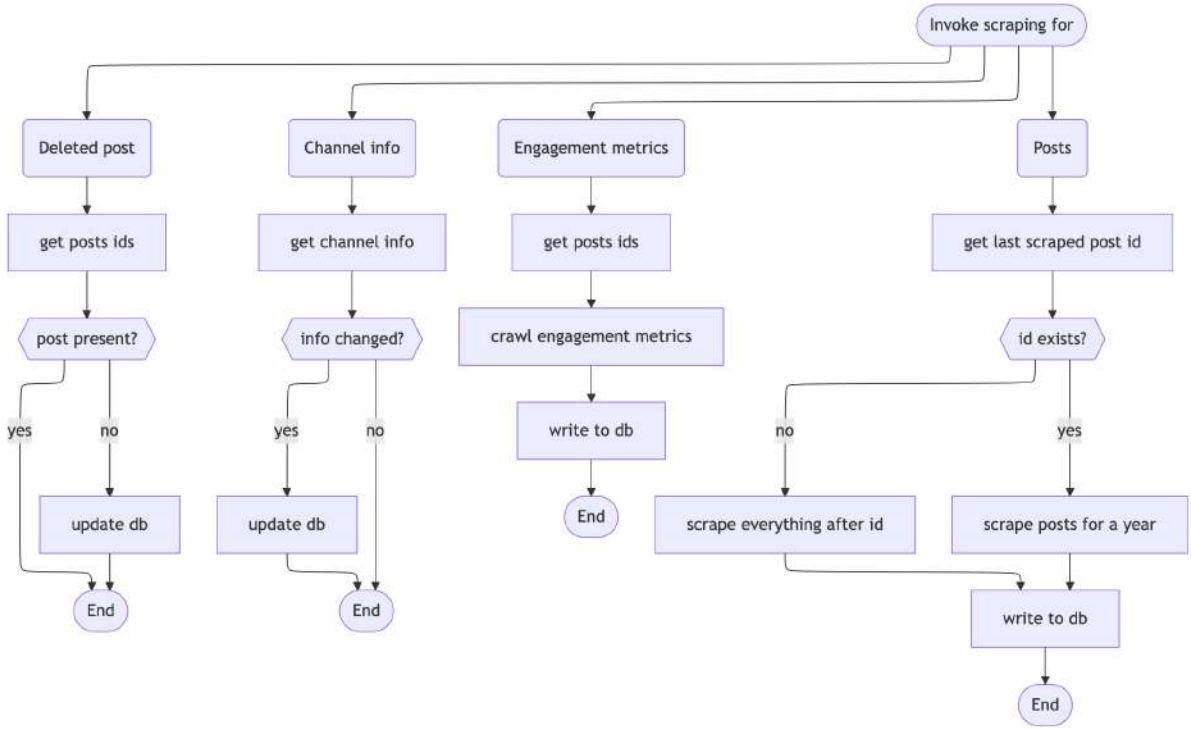

Figure 3.1: Scraping workflow

Database Schema. During initialization, the service applies migrations to create the normalized tables `channels`, `posts`, and `engagements`. Each table captures all relevant fields returned by the Telegram API, thereby enabling incremental updates without data duplication.

Concurrency and Reliability. To maximize throughput and resilience, the Scraper employs Go's native concurrency primitives—goroutines, buffered channels, and worker pools—to parallelize crawling tasks. The `Store` interface abstracts all database interactions, preventing accidental misuse and ensuring thread-safe operations. Furthermore, the crawler client is monitored via health checks that verify connectivity to both the Telegram API and the database.

Overall, this design provides a robust, scalable foundation for continuous data ingestion from Telegram channels, suitable for large-scale analytics and real-time monitoring applications.

3.2.2 Aggregator

Initialization. Aggregator is a service responsible for interacting with the database to aggregate scraped data. It runs as a standalone process, executing its aggregation task

once every hour. On the very first run it processes all data scraped over the past year; on every subsequent run it processes only the data scraped in the last hour.

Database Schema. When the service starts, it applies any pending database migrations and creates any missing tables. Once the connection to the existing database is established, the aggregator checks for the scraper's tables and then creates four additional tables if needed: `post_stats`, `word_frequency`, `bigrams` and `tf_idf_scores`.

Aggregation. Each hourly aggregation task proceeds in three stages. First, it computes post statistics by counting how many posts each channel has provided during the period, summing engagement metrics such as total views and forwards, and calculating the average post length. Second, it performs unigram counting: although simple, this step reveals which single words were most frequent. Third, it computes bigram statistics: since pairs of words carry more context than single terms, bigrams often yield deeper insights for analysis.

Term Frequency-Inverse Document Frequency. Finally, the service calculates the TF-IDF (Term Frequency-Inverse Document Frequency) scores for both unigrams and bigrams. Term frequency measures how often a term appears in a document (usually normalized by document length), but because we do not have documents we consider "day" as our document, while inverse document frequency down-weights very common terms across the entire corpus. Together, TF–IDF highlights terms that are both frequent in a given document and distinctive across the corpus, improving the relevance of downstream analyses.

3.2.3 Dashboard

Term "Dashboard" first appeared in the early 1800s, referring to the protective board on a carriage designed to keep mud from being flung up by the horses. With the rise of the automobile, the term was adopted for the instrument panel that displays the status of various vehicle systems to the driver[11].

Today, the word "Dashboard" has many definitions, ranging from "purposely created data screen"[12] to "arranged single visual display"[13]. Despite these variations, all dashboards share the goal of visualizing information: they act as a "lens" through which users can view large amounts of high-value data at a glance. By abstracting details into clear visual elements, dashboards help users focus on key metrics without being overwhelmed.

To ensure a dashboard is both clear and visually appealing, two main considerations are essential[11]:

1. **Selecting the right data.** To identify the "right" data, ask two questions—"What?" and "Why?":
   - *What do we collect?* — the text of Telegram posts.
   - *Why do we collect it?* — to analyze trends, detect correlations with real-world events, and predict future tendencies.
2. **Choosing the right visualization.** Select chart types and layouts that minimize the time required for users to interpret the data.

By combining carefully chosen data with effective visual techniques, a dashboard can guide users to insights quickly and efficiently, making it an invaluable tool for real-time monitoring and informed decision-making.

Grafana

Grafana is an open-source platform for querying, visualizing, and alerting on data from multiple sources. It can connect to time-series databases (such as Prometheus or InfluxDB), SQL or NoSQL systems, cloud monitoring services, and custom data stores via plugins. Users build dashboards by arranging panels—graphs, tables, heatmaps, and more—each driven by configurable queries and time ranges.

Its interactive interface supports templating and variables, so a single dashboard can adapt to different data sets or environments. Grafana Live adds real-time streaming capabilities, while built-in alerting lets you define thresholds and notification channels (email, Slack, PagerDuty, etc.). This combination of visualization and alerting in one tool simplifies monitoring workflows. Example of Grafana dashboard design with multiple visualizations is shown in Figure 3.2

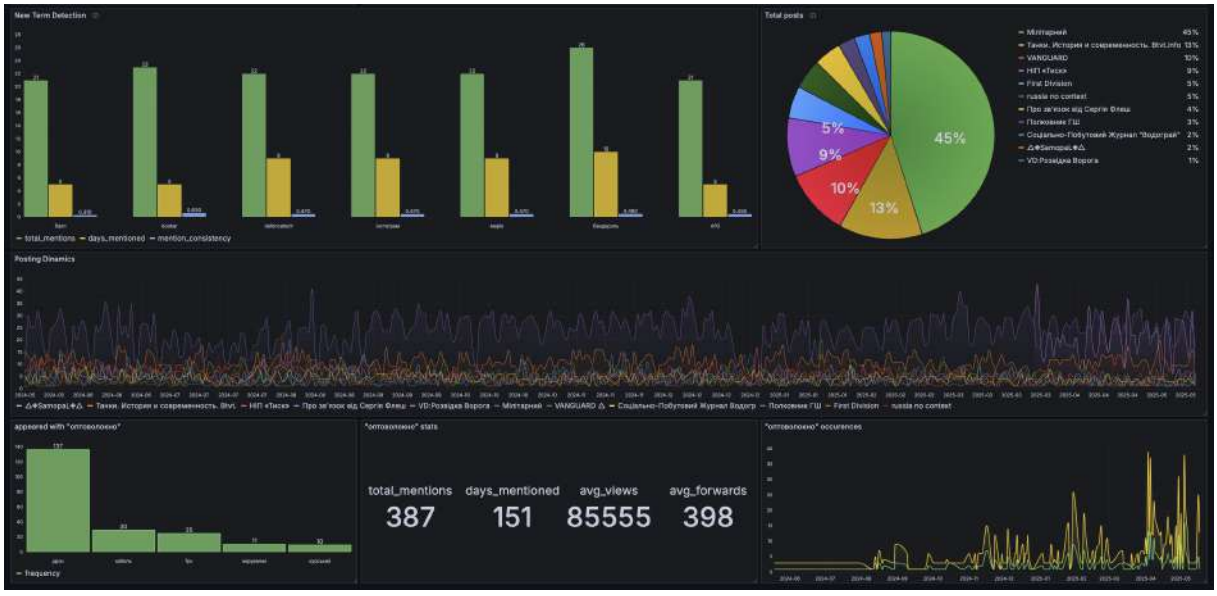

Figure 3.2: Grafana dashboard example

A Grafana dashboard is composed of panels that present data as interactive graphs, charts, and other visualizations. Each panel is powered by components—such as the data source plugin and query engine—that transform raw data into visual elements. This workflow routes data through three stages: the plugin, the query, and an optional transformation step. The figure below shows these stages, and the following sections explain each stage's purpose, usage, and significance.

Because Grafana is free under an open-source license and has a large community, it benefits from frequent updates, a wide plugin ecosystem, and extensive documentation. For this project, Grafana was chosen to display Telegram scraping metrics—post counts, engagement statistics, word frequencies, bigrams, and TF–IDF scores—in real time, also the great flexibility of grafana allows us to showcase and analyse interactions between channels: how they repost each other's posts.

3.3 Cloud Services

In this work, we examine cloud platform deployment options by evaluating the three market leading providers: Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure (Azure). As of Q2 2024, their global market shares were 32%, 23%, and 12%, respectively[14].

We compared their free-tier offerings, ecosystem features, and platform strengths:

- **AWS** offers a Free Tier covering over 100 services, including 12 months of free usage for selected products, always-free offers for certain services, and free trials for others. It also provides extensive online documentation, estimated at over 800,000 pages, and a large global user community, easing access to tutorials and support[15].

- **GCP** grants new users a 90-day free trial with $300 in credits for all services, plus always-free usage limits for core offerings such as Compute Engine, Cloud Storage, and BigQuery.
  GCP features strong container support through Google Kubernetes Engine (GKE) and provides integrated developer tools like the Cloud SDK and Cloud Build to streamline development workflows[16].

- **Azure** combines both models by offering 12 months of free access to more than 65 services, along with $200 in credits for the first 30 days.
  Azure integrates deeply with Microsoft development tools such as Visual Studio, Visual Studio Code, and GitHub, and supports hybrid cloud scenarios via services like Azure Arc and Azure Stack[17].

Each provider also has unique advantages: AWS leads in breadth of services and community support; GCP excels in container management and data analytics; Azure's

strengths lie in enterprise tool integration and hybrid cloud solutions.

3.4 Amazon Web Services

We chose Amazon Web Services as the deployment platform for our system. AWS provides the broadest range of services—over 200 products covering compute, storage, networking and managed databases—which lets us implement all required components within a single ecosystem. Its Free Tier offers 12 months of free usage for key services, reducing early-stage costs and simplifying testing. With a global infrastructure spanning 26 regions and 84 availability zones, AWS ensures low latency and high availability. Finally, AWS holds extensive security and compliance certifications (ISO, SOC, PCI DSS, GDPR) and benefits from a large user community and detailed documentation, all of which streamline deployment, troubleshooting, and ongoing maintenance.

3.4.1 Amazon Elastic Container Registry

Amazon Elastic Container Registry (ECR) is a fully managed Docker container registry that simplifies storing, managing, and deploying container images. It integrates directly with Amazon Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS), so user tasks and pods can pull images by simply specifying the ECR repository URI in their definitions. ECR stores images in Amazon S3, giving you 99.999999999% (11 nines) of data durability by default and automatically replicating across multiple Availability Zones for resilience.

ECR includes built-in image scanning powered by Amazon Inspector, which analyzes pushed images for known vulnerabilities and provides detailed findings. User can enforce a policy to block deployment of images with critical or high-severity issues. Lifecycle policies let user automatically expire untagged or older images, keeping his registry clean and cost-efficient without manual intervention.

Finally, ECR supports fine-grained access control through AWS Identity and Access Management (IAM). It is possible to grant roles and users permission to push, pull, or scan images, or even delegate access via cross-account roles. This tight security integration ensures that only authorized workloads can retrieve your container images.

3.4.2 Amazon Elastic Container Service (Fargate)

Amazon Elastic Container Service with Fargate is a serverless compute engine for containers that works with Amazon ECS, removing the need to provision or manage servers. User defines his task definitions — CPU, memory, networking, and IAM roles, after what Fargate launches containers in the background, automatically scaling capacity based on

demand. Service has pay-as-you-go model, which lets user focus on application development rather than infrastructure.

Fargate tasks run within a secure Amazon Virtual Private Cloud (VPC), which should be configured beforehand or during deployment. User is able to use AWS Secrets Manager or Parameter Store to provide credentials securely. Integration with Amazon CloudWatch provides logs and metrics out of the box, while AWS X-Ray can trace requests through deployed microservices. Fargate also supports spot capacity, letting you run tasks at a discount when spare capacity is available.

Under the hood, Fargate isolates each task in its own kernel, providing strong security boundaries between workloads. User can mix Fargate and EC2 launch types in the same ECS cluster, giving flexibility to optimize for cost or control. Task placement and scaling policies work the same across both modes, keeping deployment workflows consistent.

3.4.3 Amazon EventBridge

Amazon EventBridge is a serverless event bus that connects application data from user's apps, AWS services, and SaaS providers. It is possible to create event buses to collect events, and rules to filter or transform those events before routing them to targets such as Lambda functions, ECS tasks, or Kinesis streams. This decouples event producers from consumers and lets user build modular, event-driven architectures.

EventBridge supports scheduled events, so cron-style tasks could be trigger on schedules without a separate scheduler. The schema registry and discovery feature automatically infers event schemas and makes them available as code bindings, speeding up development by generating typed models in languages like Java or Python. It is also possible to upload custom schemas or share them across AWS accounts.

For reliability, EventBridge retries failed deliveries three times with exponential backoff and can dead-letter events to Amazon Simple Queue Service (SQS) or Amazon Simple Notification Service (SNS) if all retries fail. Also EventBridge provides possibility to configure Dead-Letter Queues (DLQs), special system to temporary store messages that can not be procedet due to some error, per rule, giving you full control over error handling. With integrations to over 120 AWS services and popular SaaS apps, EventBridge acts as the central hub for complex, scalable event workflows.

3.4.4 Amazon Relational Database Service

Amazon Relational Database Service (RDS) for PostgreSQL is a managed database that handles provisioning, patching, backup, and recovery. It provides functionality to choose instance classes (CPU and memory) and storage (General Purpose SSD or Provisioned IOPS) based on workload's performance needs. Automated backups and point-in-time recovery protect valuable data, while maintenance windows allow non-disruptive updates.

For high availability, RDS offers Multi-AZ deployments, which create a synchronous standby in a different Availability Zone and automatically fail over in case of primary instance failure. Read replicas let user scale out read traffic by creating asynchronous copies of database—useful data for reporting or analytics workloads that demand high read throughput.

Monitoring and tuning are built in: Amazon CloudWatch metrics track CPU, memory, I/O, and connections, while Performance Insights visualizes SQL query performance and database load. Service provides possibilities to adjust parameters via parameter groups and manage security through VPC isolation, IAM authentication, and encryption at rest and in transit.

3.4.5 Amazon Managed Grafana

Amazon Managed Grafana is a fully managed, secure data visualization service built on the open-source Grafana platform, which eliminates the need to provision, configure, or maintain Grafana servers yourself. It provides logically isolated *workspaces* — each a dedicated Grafana server instance, so user can create dashboards and visualizations immediately after workspace creation.

Each workspace integrates natively with AWS data sources, such as Amazon CloudWatch, Amazon RDS, other Amazon Managed Service for Prometheus, AWS X-Ray, Amazon Timestream, and Amazon OpenSearch Service, also as well as with many open-source and third-party plugins. This lets user query, correlate, and visualize metrics, logs, and traces from multiple accounts and Regions in a single pane of glass.

For user authentication and access control, Amazon Managed Grafana supports AWS IAM Identity Center and SAML 2.0 identity providers. When user enables IAM Identity Center, AMG automatically activates AWS Organizations and provisions the necessary IAM roles, simplifying multi-account access management if needed.

Security is enforced via AWS IAM policies, fine-grained workspace permission settings, and audit logging through AWS CloudTrail. User can restrict which users or groups can view, edit, or administer each workspace and data source, ensuring compliance with corporate governance requirements.

Amazon Managed Grafana offloads operational tasks—automatic scaling of compute and storage, version updates, and security patching, so that workspaces remain performant and up to date without manual intervention. Under the hood, AMG uses multi-AZ deployments and Amazon EKS control planes to provide high availability and fault tolerance.

Existing self-managed Grafana users can migrate dashboards, plugins, and data sources with minimal changes, leveraging the AMG migration tool and API. For organizations requiring enterprise features such as enhanced collaboration, enterprise plugins, and sup-

port user can upgrade any workspace to Grafana Enterprise directly within the AWS Console.

To sum up Amazon Managed Grafana delivers the flexibility and extensibility of Grafana combined with AWS's scalability, security, and operational simplicity, making it an ideal choice for real-time monitoring and visualization of complex cloud-native environments.

3.5 Terraform

To bring all our services together in one aesy-to-user way we considered usint infrastructure-as-code tool Terraform. Terraform is an open-source tool that lets user declare AWS resources in human-readable configuration files. User has to configure the AWS provider with credentials and region, then define resources like `aws_ecr_repository`, `aws_ecs_cluster`, `aws_eventbridge_rule`, and `aws_db_instance`. Running `terraform init` downloads provider plugins, and `terraform apply` creates or updates resources to match given configuration.

Terraform maintains a state file that tracks real-world infrastructure, enabling it to compute diffs and perform safe, incremental changes. Modules allow encapsulation and reuse common patterns, such as a standardized VPC or logging setup across multiple environments. User can version modules in a registry or in his own repositories for team sharing and governance.

By using variables and workspaces, Terraform supports multiple environments (development, staging, production and others) from the same codebase. The plan/apply workflow ensures changes are reviewed before execution, and the `taint` command lets you mark resources for forced replacement. With its declarative approach, Terraform delivers predictable, repeatable infrastructure provisioning.

Chapter 4

Solution Methodology

4.1 System Architecture Overview

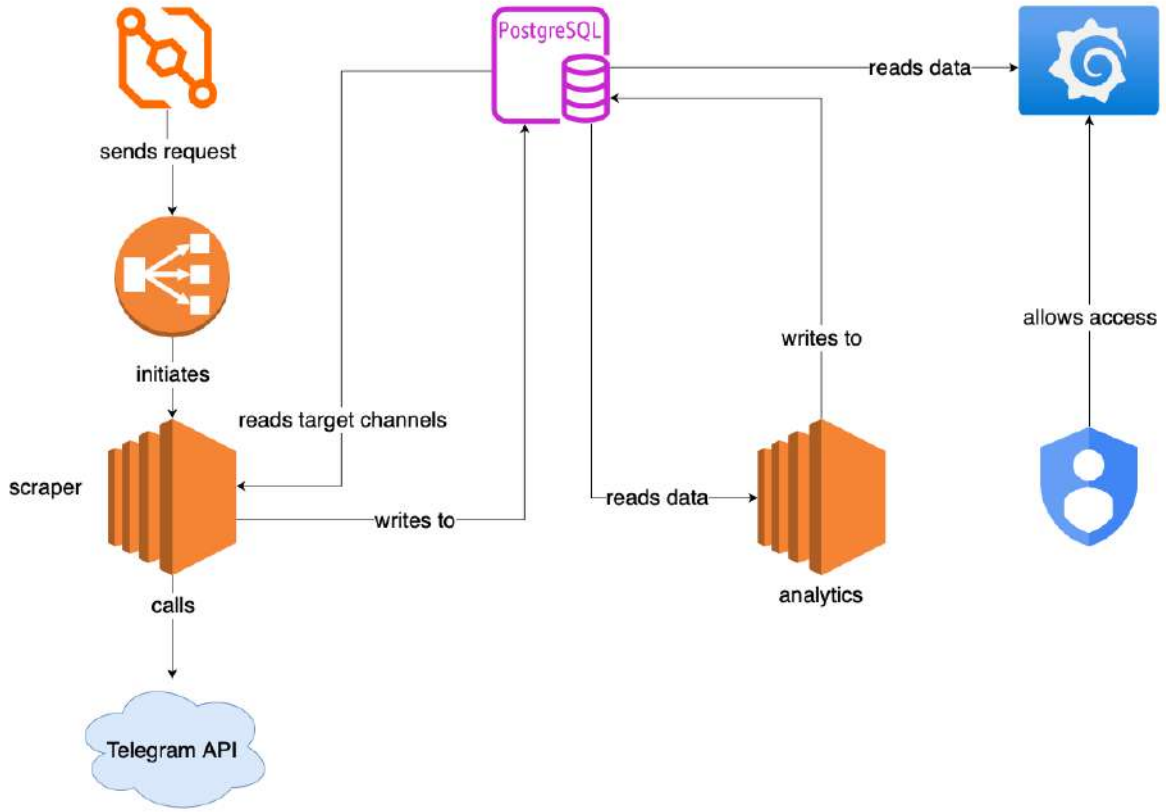

Figure 4.1: Architectural Design of the System Deployed on Amazon Web Services (AWS)

While the system is designed for local execution via Docker containers, assisting in development and testing in isolated environments, this work also provides Terraform scripts for deployment on Amazon Web Services. This cloud-based deployment strategy was chosen to achieve scalability and reliability. The system architecture is inherently modular, comprising three core components, each with it's own, distinct responsibility. The *Scraper*

Service, implemented in Go, is responsible for acquiring raw information. The *Analysis Service*, developed in Python, processes this information to derive insights. Finally, the *Visualization Dashboard*, utilizing Grafana, presents these insights to the user, allowing him to create their own visualizations for more flexible insights. Communication and data exchange between these distributed services are performed via RDS PostgreSQL database. This architectural choice promotes loose coupling, allowing components to evolve independently and be independent, Scaper don't need to wait for Analytics service to end it's calculations and vice versa.

The operational flow begins with the Scraper Service, which programmatically interfaces with the Telegram API to retrieve real-time updates from specified channels. Upon receipt, it processes these incoming messages to extract relevant data points and subsequently stores raw, unprocessed data in an Amazon RDS PostgreSQL database instance, chosen for its relational structure and reliability. The Analysis Service reads data from database and retrieves new messages. Upon retrieving a messages, it performs calculations like word's frequency, posts' generalized statistics, bigrams and tf-idf scores. Finally, the Grafana dashboard connects directly to this RDS instance, using pre-created user without permissions to change database, only read, executing queries to fetch the processed data and render dynamic, real-time visualizations, providing users with immediate insights into the analyzed content.

Network security is enforced through AWS Security Groups. These act as virtual firewalls, controlling inbound and outbound traffic for each service individually, which are isolated within their own subnets to limit the potential security breaches. Inbound traffic rules are defined with strict adherence to the principle of least privilege; for instance, only the Load Balancer (LB) is permitted to establish connections with the Scraper service. The LB is responsible for distributing incoming HTTP(S) requests across available service instances, thereby enhancing availability and responsiveness, and also provides crucial TLS (Transport Layer Security).

4.2 Deployment Strategy and Infrastructure Provisioning

The deployment of the entire AWS infrastructure is automated using Infrastructure-As-Code tool **Terraform**. This approach is fundamental to ensure that deployments are repeatable, version-controlled, and less prone to manual user-error. A primary Terraform module establishes the foundational network topology, defining the Virtual Private Cloud (VPC) along with its constituent public and private subnets, and configures the overarching Security Groups that handle network traffic flow between these subnets and services.

After establishing networking, Terraform modules instantiate AWS services required

by the system, each of them in it's own module to achieve flexible configuration. Database is configured with automated backups to prevent data loss and Multi-AZ (Availability Zone) replication to ensure high availability and resilience against single data center failures. To manage the containerized applications, **Amazon Elastic Container Registry (ECR)** is used as a private Docker image repository, hosting the Docker images for both the Go-based ingestion service and the Python-based analysis service. The services themselves are run using **Amazon Elastic Container Service (ECS) with AWS Fargate**. Fargate is a serverless compute engine for containers, abstracting away the underlying server infrastructure and allowing the services to run based on defined CPU and memory requirements, with auto-scaling capabilities to adjust capacity in response to load variations. User requests and external API calls to the ingestion service are managed by an **Load Balancer (LB)**, which routes HTTP(S) traffic to the appropriate container instances and provides a stable, singular endpoint for the service. Finally, monitoring and observability are achieved through **Amazon CloudWatch**, which logs performance metrics from all deployed resources.

4.3 Technology Stack and Rationale

The **Scraper Service** was implemented in **Golang (Go)**. This choice was primarily driven by Go's exceptional native support for concurrency, through goroutines and channels, which is highly beneficial for handling potentially high-throughput data and rapid increase in workload. Furthermore, Go has low memory footprint and efficient compilation result in lightweight, fast-starting binaries, making it an ideal candidate for a service focused on I/O operations and rapid scaling.

For the **Analysis Service**, **Python** was selected due to its mature and extensive ecosystem of libraries dedicated to text processing and computations. Libraries such as NLTK, pymorphy3 and numpy provide pre-built functionalities and robust tools that significantly accelerate the development of such systems. Python's readability and large developer community also contribute to faster development cycles and easier maintenance.

The entire AWS environment is managed using **Terraform**. Its adoption provides declarative Infrastructure as Code (IaC) capabilities. This allows the entire cloud infrastructure be defined in configuration files, which can then be version-controlled, reviewed, and reproduced across different environments or accounts with minimal manual intervention.

To ensure consistency and isolation across development, testing, and production environments, **Docker** was used for containerization. Both the Go scraper service and the Python analysis service, along with their respective dependencies, are packaged into Docker images. This approach encapsulates the application and its runtime, simplifying deployment and mitigating issues related to environment discrepancies.

Finally, for data visualization, **Grafana** was selected. Its powerful and flexible dashboarding capabilities, combined with its native support for querying PostgreSQL databases, make it an excellent choice for presenting real-time insights.

4.4 Design Considerations and Operational Constraints

The system's design was shaped by several critical factors and operational constraints that required careful consideration to ensure robust and efficient operation.

A primary constraint was the need to operate within **Telegram API rate limits**. The Telegram API imposes restrictions on the number of requests an application can make within certain timeframes. To prevent service disruption from exceeding these limits, the scraper service implements strategies such as backoff for retrying failed requests and limits on number of messages requested, ensuring respectful API interaction.

Cost optimization also played a significant role in resource selection and configuration. Decisions regarding EC2 instance classes (if applicable, though Fargate is serverless), Fargate task sizing (CPU and memory), database instance types, and storage provisioning were made to strike a balance between achieving the required performance and adhering to the budget constraints. The system is designed to allow auto-scaling mechanisms and allowing resources to scale up to meet demand and scale down during periods of low activity.

Chapter 5

Evaluation

5.1 Evaluation Methodology

The aim of this chapter is to evaluate implemented system on several points. We will evaluate resources used for scraping and aggregating history data from *01/01/2023* to *05/25/2025*, such as memory usage and speed of scraping and aggregating.

Then we perform a case study on a single term. The keyword реб was selected because it occurs frequently enough to yield statistically meaningful counts and is directly relevant to the work's focus on military-technology trends. We will compare occurrences between professional sources, Google Trends and in our implemented system.

5.2 Solution Evaluation

5.2.1 Performance evaluation

Measurement set-up. A *cold-start* run was executed that retrieves the full catalogue from *01/01/2023* to *05/25/2025* from Telegram channels. Wall-clock time was measured with the built-in logger, and peak resident–set size (RSS) was recorded during each process execution. Only posts that contain *non-empty textual content* were accepted, yielding the 78 443 documents used throughout this thesis[1].

Stage	**Runtime**	**Throughput(avg)**	**Peak RSS**
Scraping	48 min 42 s	26.8 posts/s	74.75 MB
Aggregation	28 min 20 s	46.1 posts/s	703.41 MB

Table 5.1: Performance metrics for the initial three-year backfill

[1]Forward-only posts, media-only posts and service messages are ignored.

Runtime. Scraping completes in 48:42 and aggregation in 28:20, giving a combined backfill duration of 77:02. At 26.8 and 46.1 processed posts per second respectively[2].

Memory footprint. The scraper's peak Resident Set Size (RSS) is modest (< 75 MB), corresponding to 0.98 kB per post . The aggregator, by contrast, peaks at 703 MB (9.2 kB per post). This higher figure is expected because the job materialises token lists, bigram maps and intermediate hash tables before emitting bulk inserts. Even so, the process remains comfortably inside the 2 GB memory limit of the smallest AWS `t3.small` instance.

Summary. On a cold start, the scraper collected 78 443 qualifying messages in only 48:42, after which the aggregator completed its token, bigram materialisation and calculations in another 28:20. The total wall clock time of 77:02 ($\approx$ 1:28 h).

Resource consumption also remains economical. During scraping, memory usage never exceeded 75 MB, while the more compute-intensive aggregation phase peaked at 703 MB, which is still below the 2 GB ceiling of an AWS `t3.small` instance. These results demonstrate that the proposed system is both time-efficient and cost-efficient: a single low-cost virtual machine can retrospectively populate the database with multi-year historical data within a few-hour maintenance window.

5.2.2 Case Study: "реб" (eng.: "electronic warfare")

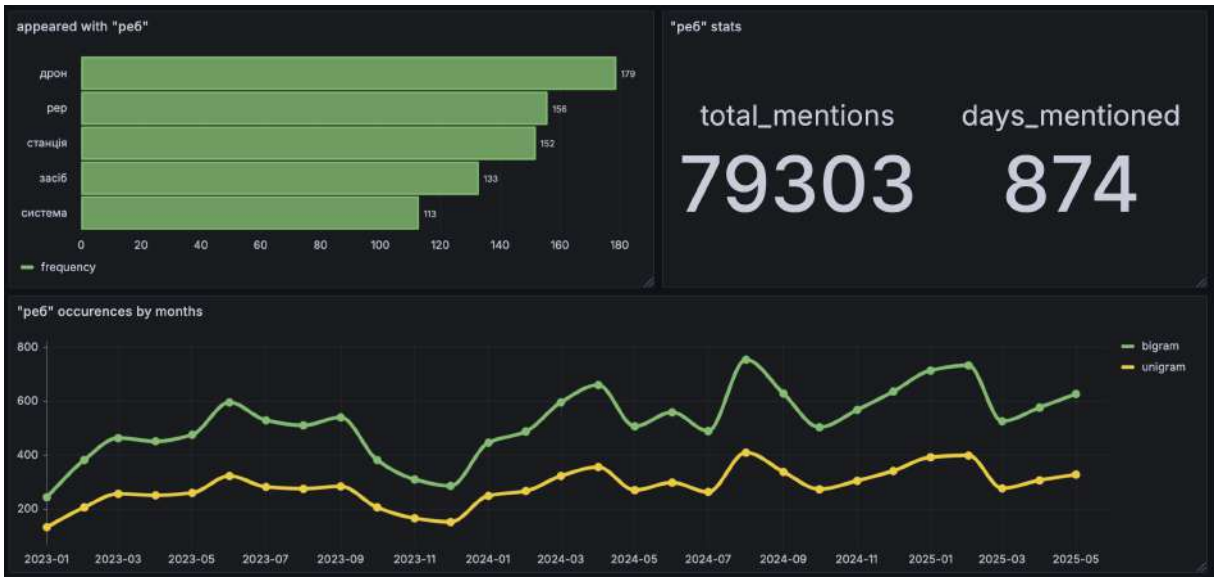

Figure 5.1: Dashboard visualizations regarding term *"реб"*

Figure 5.1 summarizes three complementary views of our aggregated metrics for the keyword "реб".

[2]Average time includes Telegram-API requests' restrictions and mandatory timeouts.

Descriptive statistics. Over the full interval, **79 303** unique posts mention «реб» on **874** distinct days. Because `days_mentioned` is an absolute count, the ratio 79 303/874 $\approx$ 90.7 gives the *mean daily density* of mentions on days when the term appears.

Co-occurrence profile. The upper-left bar chart displays the top five tokens co-occurring with «реб»:

- "дрон" (eng.: "drone") — 179 joint mentions;
- "рер" (eng.: "electronic signals intelligence") — 156 joint mentions;
- "станція" (eng.: "station") — 152 joint mentions;
- "засіб" (eng.: "means") — 133 joint mentions;
- "система" (eng.: "system") — 113 joint mentions.

Temporal dynamics. The lower-left panel plots monthly bigram counts (green) and unigram counts (yellow). Both series begin in January 2023 at roughly 240 (bigrams) and 130 (unigrams), climb to local peaks of 600/330 by June 2023, then decline to their lowest late-2023 values (280 bigrams, 150 unigrams). From early 2024 onward, activity surges again: bigrams rise to 660 in April 2024 and reach a maximum of 760 in August 2024 before dipping and peaking once more (720 in February 2025); unigrams mirror this pattern with peaks near 350–410 in spring and late 2024, settling around 320 by May 2025. These fluctuations indicate recurring waves of discussion intensity around electronic-warfare topics.

Benchmarking against external information channels. To assess timeliness, we compare our dashboard's signal with two routine sources:

1. The editorial archive of *Militarnyi*.
2. The "Google Trends" platform.
3. The dashboard from our system.

Militarnyi. The earliest mention of «реб» appears in the article "Ukrainian military received nine H10 Poseidon Mk II UAVs", published on January 15, 2023 [18], which describes delivery of drones equipped with EW protection, Israeli optics, and a thermal imaging camera.

Google Trends. Figure 5.2 overlays the normalized public-search interest for «реб» (purple) against our system's unigram counts (cyan). Google Trends remains below 50 until mid-2023, peaks at around 140 in May 2023, then gradually declines toward 80 by

mid-2025. In contrast, our Telegram-based unigram series starts near 130 in early 2023, rises to a maximum of about 350 by April 2024, and thereafter stays above roughly 250, indicating a consistently stronger and earlier signal in private-channel data.

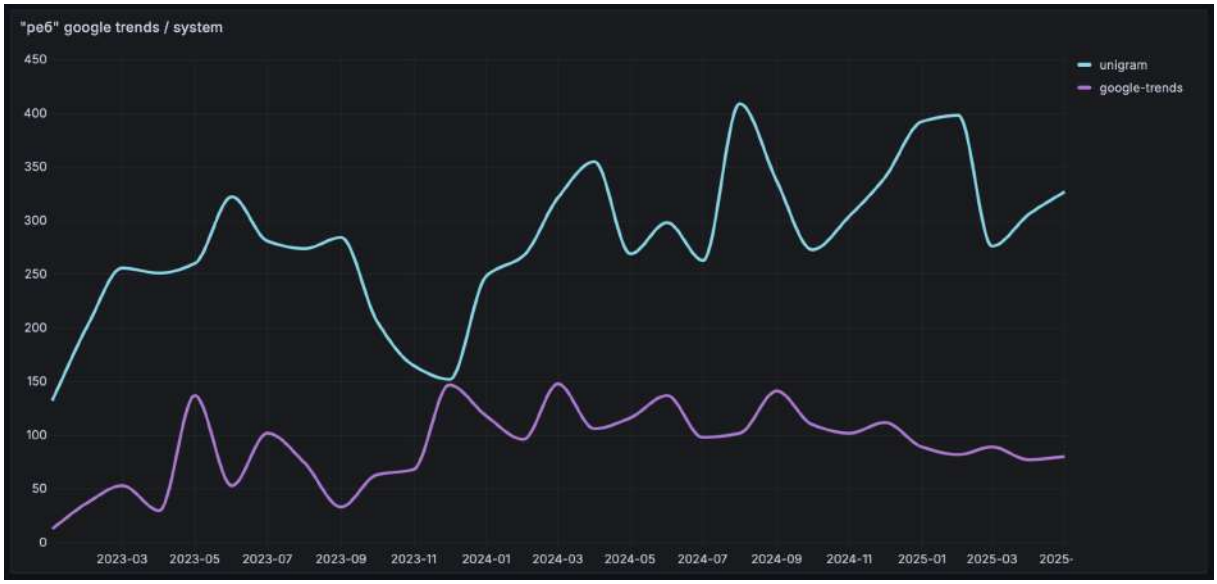

Figure 5.2: Comparison of Telegram-system unigram counts (cyan) against Google Trends (purple) for *"реб"*.

Implemented System. Our dashboard logs the first *"реб"* mention in January 2023—several months before the Google Trends uptick and before any coverage in the specialist outlet.

Summary. Telegram channels detected «реб» in January 2023, well ahead of the Google Trends surge in mid-2023 and of the Militarnyi article. The disparity in both timing and magnitude underscores that private-channel monitoring can yield an earlier and richer signal of emerging electronic-warfare discussions than either public search behavior or domain-specific news archives.

Chapter 6

Further work and Conclusions

6.1 Conclusions

This thesis addressed the challenge of monitoring and analyzing rapidly evolving trends in military technologies, particularly within the dynamic environment of Telegram channels. The manual processing of the sheer volume of data from these sources is impractical and prone to delays. The primary objective was to investigate the usability of design, and implement an automated news monitoring system tailored for this specific domain.

The research has led to the development of a modular, scalable system. This system comprises a Go-based Scraper service for data acquisition from Telegram using TDLib, a Python-based Analysis service for processing and deriving insights (such as n-grams and TF-IDF scores), and a Grafana dashboard for versatile data visualization. The system's infrastructure is defined using Terraform for reproducible deployment on Amazon Web Services, leveraging services like ECS with Fargate, RDS, and Managed Grafana to ensure reliability and scalability.

The key contributions and differentiators of this work, when compared to existing general-purpose monitoring systems and related research, are evident in several areas. Firstly, the system offers a specialized focus on military technology trends with an emphasis on Telegram channels, a platform chosen for its demonstrated rapidity and comprehensiveness as a source for up-to-date miltech information[9, 10]. This specificity contrasts with general monitoring tools and is significant given Telegram's utility as an early indicator in this domain, as underscored in the evaluation.

Furthermore, this work provides a fully automated solution in a domain often reliant on manual efforts, built with a modern, scalable cloud-native architecture, containerization, serverless compute with AWS Fargate, and Infrastructure-as-Code. This approach ensures efficiency and adaptability, marking a practical advancement. The deliberate choice of Telegram's TDLib API for comprehensive data scraping, combined with analytical components like engagement metrics and TF-IDF scores. Lastly, the research culminated

in a fully developed and deployed system, with a performance evaluation confirming its efficiency in processing substantial historical data.

In conclusion, this research successfully demonstrated the feasibility and utility of developing a specialized monitoring system for military technology trends on Telegram. The implemented solution offers a valuable automated tool for researchers, analysts, and organizations requiring timely and actionable insights from this increasingly important information source. The architectural choices and specific focus on early trend detection within the miltech domain distinguish this work from more generalized news monitoring systems.

6.2 Further work possibilities

The developed system provides a robust foundation for monitoring military technology trends on Telegram. Future research and development could further enhance its capabilities and broaden its applicability in several key areas. Expansion of data sources is a primary consideration; while Telegram is crucial, integrating data from other relevant platforms like specialized defense forums, niche miltech news websites, or other social media could provide a more versatile view. This could also involve developing capabilities for robust cross-lingual monitoring and analysis, as military technology discussions are global in nature.

Further advancements could be achieved by incorporating more sophisticated Natural Language Processing (NLP) techniques beyond the current n-gram analysis and TF-IDF scores. This includes exploring advanced topic modeling to automatically discover and track thematic shifts, implementing Named Entity Recognition (NER) specifically trained for military equipment and technologies, and deploying sentiment analysis to gauge perceptions towards specific trends. Additionally, event extraction to catalogue significant developments and relation extraction to identify connections between entities, such as a technology's use by a specific unit, would add considerable depth.

Finally, addressing the challenge of disinformation within the monitored content represents an important avenue for future research, potentially involving the analysis of linguistic patterns, source reliability, and propagation networks. More extensive longitudinal case studies across a wider array of military technologies and over longer periods would also be beneficial to further validate the system's effectiveness in early trend detection and analysis.

Bibliography

[1] Telegram Messenger Inc. *Telegram: Secure Messaging for Everyone*. URL: https://telegram.org (visited on 04/08/2025).

[2] T. Khaund, M. N. Hussain, M. Shaik, and N. Agarwal. "Telegram: Data Collection, Opportunities and Challenges". In: *Information Management and Big Data*. Communications in computer and information science. Cham: Springer International Publishing, 2021, pp. 513–526.

[3] R. Steinberger, B. Pouliquen, and E. V. D. Goot. "An introduction to the Europe Media Monitor family of applications". In: *ArXiv* abs/1309.5290 (2013).

[4] Grafana Labs. *Grafana: The open observability platform*. URL: https://grafana.com/ (visited on 04/08/2025).

[5] N. Panagiotou, A. Saravanou, and D. Gunopulos. "News Monitor: A framework for exploring news in real-time". en. In: *Data (Basel)* 7.1 (Dec. 2021), p. 3.

[6] I. Flaounas et al. "NOAM: news outlets analysis and monitoring system". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. New York, NY, USA: ACM, June 2011.

[7] C. McIntosh, ed. *Cambridge advanced learner's dictionary*. en. 4th ed. Cambridge, England: Cambridge University Press, 2013.

[8] C.-M. Chen and C.-Y. Liu. "Personalized e-news monitoring agent system for tracking user-interested Chinese news events". en. In: *Appl. Intell.* 30.2 (Apr. 2009), pp. 121–141.

[9] Gradus Research Plus. *Changes in media consumption in Ukraine in 2024*. en. Nov. 2024. URL: https://gradus.app/en/open-reports/changes-media-consumption-ukraine-2024/ (visited on 04/13/2025).

[10] Civil Network OPORA. *Media consumption of Ukrainians: The third year of a full-scale war*. en. July 2024. URL: https://www.oporaua.org/en/viyna/media-consumption-of-ukrainians-the-third-year-of-a-full-scale-war-25292 (visited on 04/13/2025).

[11] A. Janes, A. Sillitti, and G. Succi. "Effective dashboard design". en. In: *Cutter IT Journal* 26 (Jan. 2013), pp. 17–24.

[12] S. Few. “Information Dashboard Design : The Effective Visual Communication of Data / S. Few.” In: (Jan. 2006).

[13] R. Kitchin, T. Lauriault, and G. McArdle. “Knowing and governing cities through urban indicators, city benchmarking and real-time dashboards”. In: *Regional Studies, Regional Science* 2 (Feb. 2015), pp. 6–28. DOI: 10.1080/21681376.2014.983149.

[14] Synergy Research Group. *Cloud Market Growth Stays Strong in Q2 While Amazon, Google and Oracle Nudge Higher — Synergy Research Group — srgresearch.com.* Aug. 2024. URL: https://www.srgresearch.com/articles/cloud-market-growth-stays-strong-in-q2-while-amazon-google-and-oracle-nudge-higher (visited on 05/17/2025).

[15] Amazon Web Services Inc. *Free Cloud Computing Services - AWS Free Tier — aws.amazon.com.* 2025. URL: https://aws.amazon.com/free (visited on 05/18/2025).

[16] Google LLC. *Free Trial and Free Tier Services and Products — cloud.google.com.* 2025. URL: https://cloud.google.com/free (visited on 05/18/2025).

[17] MICROSOFT CORPORATION. *Explore Free Azure Services — Microsoft Azure — azure.microsoft.com.* 2025. URL: https://azure.microsoft.com/en-us/pricing/free-services (visited on 05/18/2025).

[18] V. Kushnikov. *Ukrainian military received nine H10 Poseidon Mk II UAVs.* en. Jan. 2023. URL: https://militarnyi.com/en/news/ukrainian-military-received-nine-h10-poseidon-mk-ii-uavs/ (visited on 05/25/2025).