International Journal Software Engineering and Computer Science (IJSECS)
6 (1), 2026, 1-14
Published Online April 2026 in IJSECS (http://www.journal.lembagakita.org/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v6i1.5997.

RESEARCH ARTICLE Open Access

AutoClusterAPI: A Lightweight Backend Framework for Automated Unsupervised Clustering Pipelines

Yoppy Yunhasnawa [1*], **Atif Windawati** [2], **Toga Aldila Cinderatama** [3], **Moch. Zawaruddin Abdullah** [4], **Elok Nur Hamdana** [5]

[1*,3,4,5] Politeknik Negeri Malang, Malang City, East Java Province, Indonesia.
[2] Politeknik Negeri Semarang, Semarang City, Central Java Province, Indonesia.

**Corresponding author*: yunhasnawa@polinema.ac.id

Received: November 29, 2025; Accepted: February 2, 2026; Published: April 1, 2026.

Abstract: This study presents AutoClusterAPI, a lightweight and extensible backend system designed to simplify and accelerate unsupervised clustering workflows. The system addresses a recurring problem in data analysis practice: many practitioners need rapid clustering capabilities but lack the programming or statistical background required to build complete pipelines from scratch. AutoClusterAPI provides an automated, endpoint-driven solution that allows users to perform every stage of clustering — from data loading and cleaning to feature preparation, algorithm execution, profiling, and visualization — through standard HTTP requests. The system is built using Python and the FastAPI web framework, supports eight clustering algorithms, and includes automated preprocessing alongside PCA-based visualization. Functional testing confirms that all endpoints behave correctly under both valid and invalid inputs, establishing the reliability of the system. A case study using a customer segmentation dataset further demonstrates its practical utility, showing that AutoClusterAPI can efficiently generate meaningful cluster structures and interpretable visual outputs. The system offers an accessible yet configurable environment for rapid clustering analysis and establishes a basis for future extensions and real-world deployment.

Keywords: Clustering Pipeline; Backend Automation; FastAPI; Unsupervised Learning; PCA Visualization; Data Preprocessing; API-Driven Analytic.

1. Introduction

Artificial Intelligence (AI) has experienced rapid growth over the past decade, with its applications spreading across nearly every sector of modern society. This growth is driven by advances in data availability, computational power, and algorithmic development. A central branch of AI is Machine Learning (ML), a field that enables systems to learn patterns and make decisions from data rather than from explicit rules. Among the primary categories of ML, unsupervised learning holds a distinct position: it allows machines to discover structure within unlabeled datasets — datasets that arise naturally in many practical settings where manual annotation is either costly or simply not feasible. Within unsupervised learning, clustering is one of the most widely used techniques. Its purpose is to group data points such that items within the same cluster share higher similarity compared to those in different clusters. Clustering has proven applicable across many domains, including customer segmentation for marketing analytics (Xu & Tian, 2015), content recommendation in digital media platforms (Aggarwal *et al.*, 2016), and cybersecurity threat detection for identifying anomalous

behaviors in network logs (Chandola *et al.*, 2009). These cases illustrate clustering as a versatile analytical method for extracting structured knowledge from complex, unlabeled datasets.

Performing clustering, however, is not straightforward. For many users — particularly those from non-technical or managerial backgrounds — clustering remains difficult to execute well (Amershi *et al.*, 2014). A proper clustering workflow typically requires several sequential steps: data cleaning, feature engineering, normalization, algorithm selection, hyperparameter tuning, visualization, and interpretation. These steps demand competence in both statistics and programming (Géron, 2022), which creates real barriers for stakeholders who need results but lack technical expertise. The problem is further compounded when clustering is needed under time pressure — business meetings, audits, rapid dataset examinations, incident analyses, or executive decision-making — where clear cluster summaries and visual outputs may be required within minutes, not hours (Few, 2009).

Several tools exist to lower this barrier. KNIME Analytics Platform (Franciska & Swaminathan, 2017), Orange (Demšar *et al.*, 2013), Tableau (Yuan, 2025), Microsoft Excel (Aravind *et al.*, 2010), and IBM SPSS Modeler (Marchev, 2021) all support visual clustering workflows to varying degrees. Yet these tools share a common limitation: they are built as general-purpose analytics platforms and cannot be deeply tailored to domain-specific requirements, automated pipelines, or custom reporting formats. Users who need that level of control typically fall back on programming libraries such as Python's scikit-learn or R's statistical ecosystem (Raya-Tapia *et al.*, 2025) — which brings the accessibility problem full circle. The gap between ease of use and the need for customization remains largely unaddressed by existing solutions.

This paper introduces AutoClusterAPI, a lightweight yet configurable system designed to close that gap. AutoClusterAPI allows users to deploy a backend service that exposes a structured set of REST API endpoints covering data preprocessing, feature preparation, algorithm selection, clustering execution, profiling, and visualization. By encapsulating clustering logic within API endpoints, the system allows non-expert users to run complex clustering workflows through simple HTTP requests, while simultaneously enabling developers to build custom dashboards, automated reporting systems, and domain-specific applications on top of the API. AutoClusterAPI thus offers both accessibility and extensibility — a practical approach to operationalizing clustering workflows in environments where speed, customization, and usability must coexist.

2. Related Work

Research relevant to AutoClusterAPI spans four broad areas: clustering algorithms and their theoretical foundations, tools and platforms for accessible data analysis, backend API development with modern web frameworks, and scalability concerns in large-scale clustering systems.

2.1 Clustering Algorithms and Their Applications

Clustering has been studied extensively as a core technique in unsupervised learning. Xu and Tian (2015) provide a broad survey of clustering algorithms, categorizing methods by their underlying principles — partitioning, hierarchical, density-based, and model-based — and discussing their respective strengths across different data types. This foundational work establishes the theoretical basis for algorithm selection in automated pipelines. Among partition-based methods, K-Means remains the most widely adopted algorithm due to its computational efficiency and scalability. Ahmed *et al.* (2020) conducted a comprehensive survey and performance evaluation of K-Means, confirming its suitability for datasets with well-separated, roughly spherical cluster structures. For probabilistic clustering, Yang *et al.* (2012) proposed a robust EM-based algorithm for Gaussian Mixture Models (GMM), demonstrating that soft-assignment clustering better captures overlapping distributions compared to hard-partition methods. Density-based and hierarchical approaches address a different class of problems. Deng (2020) reviewed DBSCAN and its density-driven cluster detection mechanism, while Hajihosseinlou *et al.* (2024) demonstrated the practical utility of both DBSCAN and Mean Shift in geochemical anomaly mapping — a domain where cluster boundaries are rarely linear. Agrawal *et al.* (2016) extended this by validating OPTICS for spatio-temporal clustering, showing that variable-density environments are better handled without a fixed density threshold. On the hierarchical side, Tokuda *et al.* (2022) revisited agglomerative clustering and examined how different linkage criteria affect the resulting cluster hierarchy, while Garg *et al.* (2024) demonstrated the utility of BIRCH in resource usage prediction for cloud data centers, confirming that CF-Tree-based incremental clustering remains practical for memory-constrained environments. Tang *et al.* (2022) proposed a unified one-step spectral clustering method that reduces computational overhead while retaining the ability to capture non-linearly separable structures.

Beyond algorithm development, clustering has been applied across a wide range of practical domains. Aggarwal *et al.* (2016) discuss clustering as a foundational mechanism in recommender systems, where user and item groupings drive content personalization in digital media platforms. Chandola *et al.* (2009) survey anomaly detection methods, positioning clustering as a primary technique for identifying irregular behaviors

in network logs and cybersecurity contexts. These domain applications collectively justify the need for a general-purpose clustering backend that can be adapted across different use cases without requiring users to rebuild pipelines from scratch each time.

2.2 Tools, Platforms, and Programming Environments

A number of tools have been developed to make clustering accessible to non-programmers. Franciska and Swaminathan (2017) applied multiple clustering algorithms using KNIME Analytics Platform for churn prediction, demonstrating that visual workflow tools can lower the technical barrier for domain practitioners. Demšar *et al*. (2013) introduced Orange as a Python-based data mining toolbox with a visual interface, enabling users to construct analytical pipelines without writing code. Yuan (2025) combined K-Means with Tableau for supply chain risk management, illustrating how visualization platforms extend the interpretability of clustering results in business settings. Aravind *et al*. (2010) explored clustering directly within Microsoft Excel, while Marchev (2021) evaluated IBM SPSS Modeler for data segmentation tasks — both highlighting the persistent demand for clustering capabilities in familiar, low-barrier environments. Despite their contributions, these tools share a common constraint: they function as closed platforms that offer limited support for programmatic customization, automated pipeline execution, or domain-specific integration. Raya-Tapia *et al*. (2025) compared Python, R, and MATLAB as programming environments for clustering, acknowledging that while these languages offer full flexibility, they require substantial technical proficiency — a barrier that general-purpose tools were originally designed to eliminate, yet have not fully resolved.

2.3 Backend API Development and Stateless Architecture

The development of AutoClusterAPI draws on established practices in microservice and API design. Peralta (2023) provides a practical guide to building microservice APIs using Python, Flask, FastAPI, and OpenAPI, establishing the design principles that underpin endpoint-driven backend systems. Tragura (2022) specifically addresses FastAPI for building secure and scalable Python microservices, detailing how Python type annotations and asynchronous request handling reduce development overhead considerably. Alla (2025) further demonstrated that FastAPI can sustain high-throughput gateway operations in microservice communication, confirming its suitability for production-grade backend deployments. A critical architectural consideration in multi-user backend systems is statefulness. Kablan *et al*. (2015) examined stateless network functions and showed that removing shared in-memory state between requests significantly improves scalability, fault isolation, and concurrent session safety — a principle AutoClusterAPI adopts directly by writing all intermediate outputs to session-specific filesystem directories rather than holding them in memory.

2.4 Visualization, Scalability, and API Testing

For high-dimensional clustering results to be interpretable, dimensionality reduction is a necessary step. Greenacre *et al*. (2022) provide a thorough treatment of Principal Component Analysis (PCA), explaining how orthogonal projection onto principal components preserves maximum variance while reducing dimensionality to a manageable number of axes. Few (2009) complements this by arguing that effective visualization design directly determines whether analytical results are understood and acted upon by decision-makers — not merely a cosmetic concern. AutoClusterAPI applies PCA specifically for two-dimensional scatter plot generation, following these established principles to produce outputs that are both statistically grounded and visually interpretable. One known limitation in the algorithm set supported by AutoClusterAPI is the computational cost of Spectral Clustering. Pourkamali-Anaraki (2020) proposed Nyström approximation to reduce similarity matrix construction costs, Wu *et al*. (2018) introduced random binning features to approximate the kernel matrix at scale, Khoa and Chawla (2011) explored commute time embeddings as a memory-efficient graph representation, and Li *et al*. (2016) proposed sequential spectral clustering that processes data in segments to avoid holding the full similarity matrix in memory. These works collectively inform the dataset size constraints applied in the case study of this paper, where a 20% sample was used to ensure all eight algorithms could be executed under consistent experimental conditions. Finally, validating the correctness of RESTful endpoints requires a systematic approach. Martin-Lopez *et al*. (2021) introduced RESTest, an automated black-box testing tool for RESTful web APIs, demonstrating that input-output verification at the HTTP interface level is sufficient for confirming functional correctness without requiring access to internal implementation details — an approach directly adopted in the functional testing methodology of this study.

3. Methodology

AutoClusterAPI is implemented using the Python programming language and the FastAPI web framework (Peralta, 2023). FastAPI is widely recognized for its simplicity, high performance, readable syntax, and native support for asynchronous processing, making it well-suited for developing efficient backend services and

machine learning pipelines (Alla, 2025). With FastAPI, developers define Python functions annotated with HTTP method decorators (e.g., @app.get, @app.post), while the framework automatically handles routing, request parsing, validation, and RESTful operations such as GET, POST, PUT, and DELETE (Tragura, 2022). This approach reduces boilerplate code substantially and lowers the entry barrier for deploying machine-learning-enabled backend systems. AutoClusterAPI uses these capabilities to deliver a unified and fully automated clustering pipeline that can be triggered entirely through HTTP endpoints. When a request arrives from a client — whether a web browser, Postman, or any REST-compatible application — FastAPI receives and parses it. Based on the endpoint accessed, the AutoClusterAPI layer determines the appropriate operation and delegates the task to the corresponding static method in the Analyzer class. The Analyzer class then executes the required computational step, writes intermediate or final outputs to the filesystem, and returns structured results to AutoClusterAPI. These results are then formatted and delivered to the user as JSON, enabling straightforward connection with external applications and analytical workflows. The general system architecture is illustrated in Figure 1.

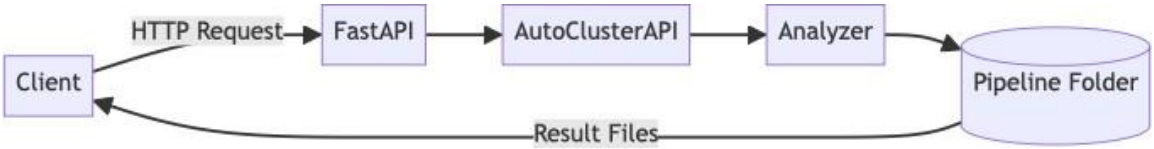

Figure 1. General system architecture

3.1 Foundation Classes

The core of the system consists of two main classes: Analyzer and AutoClusterAPI. These two classes work side-by-side to carry out the main functions of the API, as depicted in Figure 2.

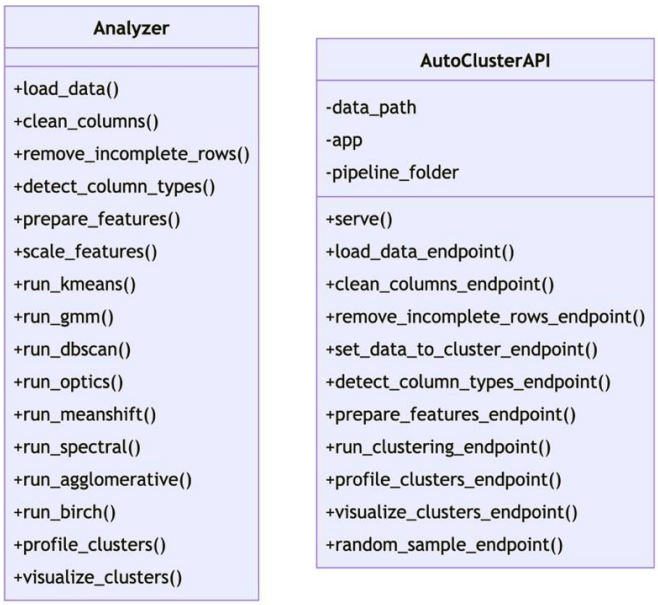

Figure 2. Main class diagram

3.1.1 Analyzer Class

The Analyzer class encapsulates all computational logic in the clustering pipeline. All methods are implemented as stateless static methods, making them compatible with backend deployment environments where requests must remain isolated across concurrent sessions (Kablan *et al.*, 2015). Because of this stateless design, intermediate outputs are not held in memory or in a database; every processing step writes its results to dedicated files stored under user-specific session directories, ensuring that each API call remains independent and unaffected by requests from other users running in parallel. The Analyzer class implements a suite of run_xxx methods that expose eight clustering algorithms, each selected for its methodological strengths and complementary analytical properties. K-Means is included as one of the most widely adopted partition-based clustering methods, offering computational efficiency and good scalability for datasets with roughly spherical, well-separated cluster structures (Ahmed *et al.*, 2020). Gaussian Mixture Models (GMM) extend this with a probabilistic, soft-clustering alternative where data points may belong to multiple clusters with associated likelihoods, enabling more nuanced modeling of overlapping distributions (Yang *et al.*, 2012). DBSCAN is included for its ability to identify clusters of arbitrary shape while simultaneously detecting noise points and outliers, making it effective in density-based scenarios (Deng, 2020). OPTICS generalizes DBSCAN

by removing the need for a single density threshold, supporting the extraction of clusters with varying densities within the same dataset (Agrawal *et al.*, 2016). Mean Shift offers a non-parametric, mode-seeking approach that identifies clusters as density peaks using kernel estimation, eliminating the requirement to predefine the number of clusters (Hajihosseinlou *et al.*, 2024). Spectral Clustering addresses non-linearly separable groups by applying graph-based eigenvalue decomposition, capturing cluster boundaries that traditional distance-based methods often miss (Tang *et al.*, 2022). Agglomerative Clustering employs a hierarchical bottom-up process that merges observations into nested cluster structures, supporting multi-level interpretation and dendrogram-based analysis (Tokuda *et al.*, 2022). Finally, BIRCH handles very large datasets efficiently by incrementally building a compact CF-Tree representation, enabling scalable clustering even in memory-constrained environments (Garg *et al.*, 2024). Beyond executing clustering, the Analyzer class provides two key supporting functions. Analyzer.profile_clusters() generates interpretive summaries by calculating the highest and lowest mean feature values for each cluster, giving users a clear picture of the defining characteristics within each segment. Analyzer.visualize_clusters() produces a two-dimensional scatter plot in PNG format using Principal Component Analysis (PCA) — a dimensionality reduction technique that transforms high-dimensional data into a smaller number of orthogonal components capturing the most significant variance in the dataset (Greenacre *et al.*, 2022). This allows clustering results involving potentially dozens of variables to be represented visually in two dimensions while retaining the essential structure of the data.

3.1.2 AutoClusterAPI Class

The AutoClusterAPI class functions as the primary interface connecting the Analyzer class with the FastAPI framework. This class contains several architectural components that collectively enable a fully automated clustering pipeline. The routing layer defines a structured set of FastAPI endpoints covering each stage of the workflow — from data loading and cleaning to feature preparation, clustering execution, profiling, and visualization — allowing users to perform analytical operations through simple HTTP requests. A dedicated session management system automatically generates unique session directories for each user, ensuring multi-user isolation and preventing interference between parallel activities. A filesystem-based pipeline controller coordinates the reading and writing of intermediate artifacts between processing stages, so that each step can be executed independently or as part of the full end-to-end workflow provided by the /auto-cluster endpoint. An algorithm dispatcher maps user-specified algorithm identifiers to their corresponding Analyzer.run_xxx() methods, guaranteeing consistent invocation across all supported algorithms. A response formatter converts intermediate results — DataFrames, cluster profiles, and PCA visualizations — into structured JSON, TXT, or HTML formats suitable for API consumption. Finally, an automated pipeline executor implements the logic behind the /auto-cluster endpoint, enabling users to run the entire clustering process from data ingestion to PCA-based visualization through a single API call. Through this modular, stateless, and file-driven design, AutoClusterAPI provides configurability for advanced users and simplicity for non-programmers, enabling fast, repeatable, and interpretable clustering workflows through a fully operational backend service.

3.2 Clustering Pipeline Endpoints

AutoClusterAPI is designed with simplicity as its foundational principle, ensuring that developers and analysts can rapidly build a complete clustering backend with minimal configuration effort. In many cases, practitioners only need a mechanism to load data, prepare features, execute clustering algorithms, and retrieve results through standardized interfaces. AutoClusterAPI addresses this need by allowing users to operationalize an entire clustering pipeline using only a small amount of Python code. To start the system, users prepare a readable CSV dataset and instantiate the AutoClusterAPI class within a short script, as shown in Listing 1.

Listing 1. Initiation code

```
from autocluster_api import AutoClusterAPI

app = AutoClusterAPI(data="data/marketing.csv")
app.serve()
```

Upon execution, AutoClusterAPI automatically exposes a set of RESTful endpoints covering each phase of the clustering process, enabling interaction through web browsers, command-line HTTP clients, or API testing tools. The overall pipeline structure is illustrated in Figure 3.

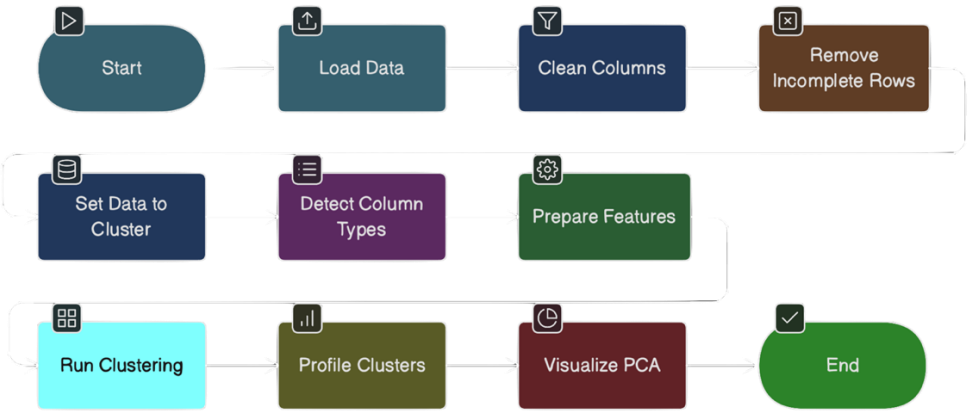

Figure 3. Clustering pipeline

The pipeline is structured as a clear sequence of staged endpoints. The /load-data endpoint initializes the process by reading the user-provided CSV using Analyzer.load_data(), respecting user-defined delimiters and preview limits. The loaded dataset is stored as step-1-loaded-data.csv and returned in JSON format, allowing the user to verify input integrity before proceeding. The /clean-columns endpoint then applies column normalization via Analyzer.clean_columns(), standardizing column names by removing whitespace, special characters, and formatting inconsistencies — an essential step before feature engineering — with the result stored as step-2-cleaned-columns.csv. The /remove-incomplete-rows endpoint performs row-level sanitization by removing entries containing null, blank, or whitespace-only values, minimizing the risk of algorithmic errors caused by missing data, and exports the result as step-3-removed-incomplete-rows.csv. The /set-data-to-cluster endpoint identifies the dataset to be used in subsequent stages — users may target Step 1, Step 2, or Step 3 outputs depending on data quality requirements — consolidated into data-to-cluster.csv as the canonical base for feature extraction and clustering.

Column classification is handled by /detect-column-types, which uses Analyzer.detect_column_types() to inspect each attribute and classify it as numerical, categorical, boolean, or text-based, storing the result as step-4-detected-column-types.json to ensure transparency and reproducibility in the feature selection process. Feature preparation is carried out through /prepare-features, where users specify arrays of numerical and categorical attributes; Analyzer.prepare_features() constructs a modeling-ready feature matrix exported as step-5-prepared-features.csv. Clustering execution is handled by /run-clustering, which loads the prepared feature matrix, triggers feature scaling and PCA projection to two dimensions, and executes the user-specified algorithm — K-Means, GMM, DBSCAN, OPTICS, Mean Shift, Spectral Clustering, Agglomerative Clustering, or BIRCH. Results are written to step-6-cluster-result-<algorithm>.csv, and silhouette scores are automatically computed to provide a quantitative measure of cluster quality. The /profile-clusters endpoint computes feature-level summaries using Analyzer.profile_clusters(), identifying dominant and least dominant attributes per cluster, with outputs stored in step-7-profiled-clusters.txt and reflected in the JSON response. The /visualize-clusters endpoint generates PCA-based scatter plots saved as step-8-visualized-cluster.png, returning the public URL for user access. The /random-sample endpoint allows users to extract a percentage-based random subset from Steps 1–3 for preliminary inspection and exploratory analysis. Finally, the /auto-cluster endpoint orchestrates all previous stages — loading, cleaning, feature preparation, clustering, profiling, and visualization — within a single API call, reflecting AutoClusterAPI's central objective: enabling rapid, reproducible clustering pipelines without requiring deep expertise in machine learning or backend development.

3.3 Clustering Flow

The sequence diagram in Figure 4 illustrates the operational workflow triggered when the /run-clustering endpoint is executed. The interaction involves four primary components — User, AutoClusterAPI, Analyzer, and the FileSystem — and captures the full lifecycle of a clustering request from input reception to result delivery.

Listing 2. Clustering endpoint

```
POST /run-clustering?algorithm=<alg>
```

The process begins when the User issues an HTTP POST request with an algorithm parameter indicating the clustering technique to be applied (e.g., k-means, GMM, DBSCAN). Upon receiving the request, AutoClusterAPI initiates the pipeline by coordinating a sequence of operations executed deterministically and statelessly.

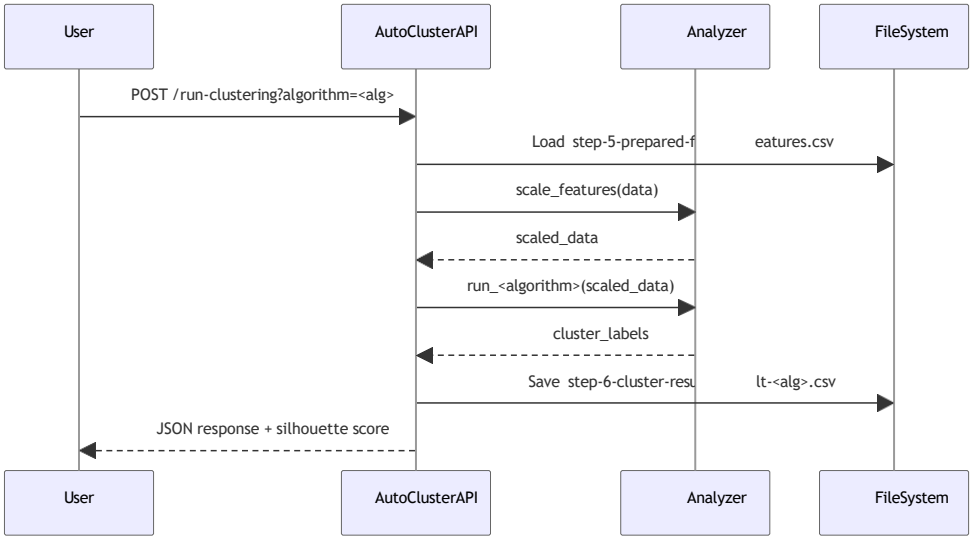

Figure 4. Clustering endpoint sequence diagram

AutoClusterAPI first retrieves the previously prepared feature dataset by loading step-5-prepared-features.csv from the FileSystem — a persistent, session-isolated storage mechanism that ensures concurrent users operate independently without shared in-memory state. The loaded data is then forwarded to the Analyzer via scale_features(data), which standardizes the features using z-score normalization to ensure consistent numerical ranges across all attributes, a necessary step for distance-based algorithms. The resulting scaled_data is returned to AutoClusterAPI, which then instructs the Analyzer to execute the selected algorithm by invoking the dynamically mapped function run_<algorithm>(scaled_data). Depending on the requested method, this may trigger run_kmeans(), run_gmm(), run_dbscan(), or other supported algorithms. The Analyzer processes the scaled dataset and generates a set of cluster_labels, each representing the assigned cluster for a corresponding observation. AutoClusterAPI then persists the output by saving it to the FileSystem as step-6-cluster-result-<alg>.csv, computes a silhouette score to assess cluster quality, and formats both the metric and cluster labels into a structured JSON response returned to the User. By combining stateless operations, file-based persistence, and modular endpoint design, AutoClusterAPI ensures that clustering workflows remain reproducible, transparent, and straightforward to connect with external applications.

4. Result and Discussion

4.1 Results

This section presents the outcomes of the backend system developed in this study, along with the results of the black-box functional testing conducted to evaluate its correctness. Clustering experiments using a customer segmentation dataset from Kaggle (vetrirah, 2025) are also reported.

4.1.1 Implementation Result

The final output of this research is a fully functional backend system that automatically provides a collection of RESTful endpoints for performing end-to-end clustering operations based on the pipeline described in the previous section. Each stage — from data loading to feature preparation, cluster generation, profiling, and visualization — is exposed through a dedicated endpoint, enabling users to orchestrate or automate clustering workflows programmatically. To support reproducibility and further development, the implementation of AutoClusterAPI has been released as an open-source project, available at: https://github.com/yunhasnawa/AutoClusterAPI. Figure 5 shows a screenshot of the automatically generated API documentation accessible via http://root-url/docs. This page is produced using Swagger UI, a built-in feature of FastAPI that renders interactive API specifications based on Python annotations. Swagger UI allows users to explore all available endpoints, understand input formats, expected outputs, parameter types, and possible server responses — significantly lowering the adoption barrier, especially among practitioners unfamiliar with backend development or those who wish to experiment with clustering workflows without writing additional client-side code. As shown in the screenshot, all endpoints provided by AutoClusterAPI —

from /load-data to /auto-cluster — are grouped and listed automatically by FastAPI, and users can interactively test each endpoint by supplying input parameters and observing the corresponding JSON responses directly from the browser without requiring external tools.

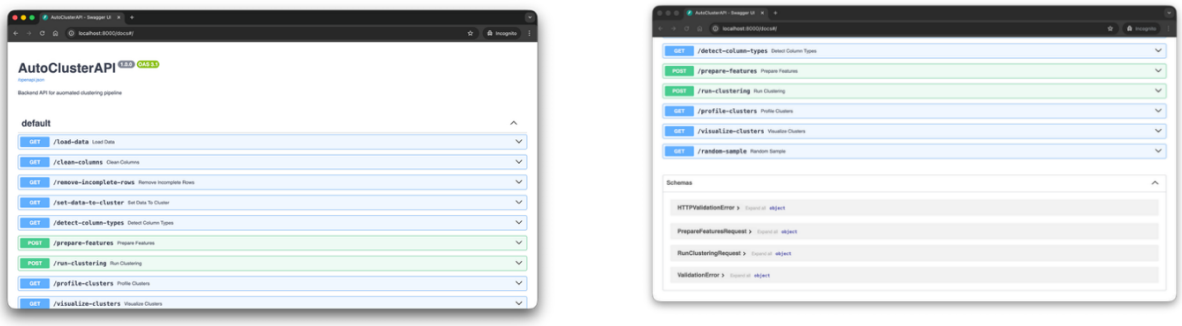

Figure 5. API documentation page

Figure 6 illustrates an example of invoking one of the API endpoints using the "Try it out" feature of Swagger UI. This feature allows users to input parameters, execute the request, and view real-time results such as processed datasets, generated cluster labels, silhouette scores, and output file paths. Such interactivity makes the system particularly suitable for rapid experimentation, demonstrations, and teaching environments. The implementation results confirm that AutoClusterAPI successfully delivers a modular and user-friendly backend environment for clustering tasks — with automatic documentation, interactive testing tools, and a standardized sequence of endpoints that offer both ease of use and extensibility across a wide range of analytical and educational applications.

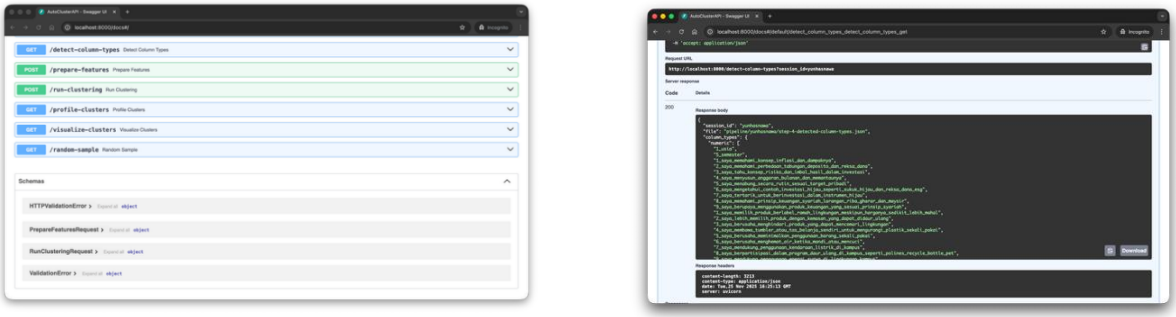

Figure 6. API try out

4.1.2 Functional Testing

Functional testing of AutoClusterAPI constitutes black-box functional verification, wherein test cases are designed solely based on the system's specification and observable behavior, without requiring insight into internal source code or architecture (Martin-Lopez *et al.*, 2021). Each endpoint is treated as a component under test: inputs (HTTP requests and parameters) are delivered, and outputs (JSON responses, generated files) are compared to expected results. This method is appropriate for AutoClusterAPI because it operates via RESTful endpoints and orchestrates a high-level data processing pipeline spanning multiple discrete stages — data loading, cleaning, feature preparation, clustering, profiling, and visualization (Pourkamali-Anaraki, 2020; Wu *et al.*, 2018; Khoa & Chawla, 2011; Li *et al.*, 2016). Evaluating input-output correctness, artifact generation, error handling, and stateless session behavior directly aligns with the system's external contract. By adopting this approach, the testing suite validates whether each endpoint fulfills its designed responsibility, isolates failures to incorrect behavior rather than implementation details, and confirms that the service is ready for use by non-expert users who interact with the API purely through its HTTP interface.

Table 1. Functional test cases and results

No	Endpoint	Test Case	Description	Input (Parameter/Body)	Expected Output	Status
1	/load-data	TC-01	Load CSV data with valid separator, limit=10	separator=";"	File step-1-loaded-data.csv saved, JSON showed 10 first lines	Success

2	/load-data	TC-02	Load CSV data without limit	separator=";"	All rows shown in JSON	Success
3	/load-data	TC-03	Use invalid separator	separator=""	Error message in JSON (file cannot be read)	Valid
4	/clean-columns	TC-04	Cleaning column names in CSV data	—	File step-2-cleaned-columns.csv contains cleaned column names (lowercase, underscore)	Success
5	/remove-incomplete-rows	TC-05	Delete rows with missing values	—	File step-3-removed-incomplete-rows.csv contains only complete rows	Success
6	/remove-incomplete-rows	TC-06	Previous file does not exist	—	JSON error: file missing	Valid
7	/set-data-to-cluster	TC-07	Set step 1 as clustered data	step=1	data-to-cluster.csv = copy from step 1	Success
8	/set-data-to-cluster	TC-08	Set step 3 as clustered data	step=3	data-to-cluster.csv = copy from step 3	Success
9	/set-data-to-cluster	TC-09	Invalid step	step=7	JSON error "step must be 1/2/3"	Valid
10	/detect-column-types	TC-10	Detect columns on data-to-cluster.csv	numeric=['age']	File step-4-detected-column-types.json saved, contains categorical columns	Success
11	/prepare-features	TC-11	Prepare features with valid input columns	numeric=['age'], categorical=['gender']	File step-5-prepared-features.csv saved	Success
12	/prepare-features	TC-12	Non-existent column	numeric=['salary']	JSON error "column not found"	Valid
13	/run-clustering	TC-13	Running kmeans	algorithm="kmeans"	File step-6-cluster-result-kmeans.csv saved + silhouette score	Success
14	/run-clustering	TC-14	Running birch	algorithm="birch"	File cluster result birch saved + score	Success
15	/run-clustering	TC-15	Unknown algorithm	algorithm="unknown"	JSON error "algorithm not supported"	Valid
16	/profile-clusters	TC-16	Profiling cluster results	algorithm="kmeans"	File txt/json/html saved in step-7	Success
17	/profile-clusters	TC-17	Unknown cluster file	algorithm="kmeans"	JSON error	Valid
18	/visualize-clusters	TC-18	Cluster visualization	algorithm="birch"	File PNG step-8-visualized-cluster-birch.png saved + URL path	Success
19	/visualize-clusters	TC-19	Wrong algorithm	algorithm="abc"	JSON error	Valid
20	/random-sample	TC-20	Sample 5% from step 1	step=1, percent=5	Shows 5% sample	Success
21	/random-sample	TC-21	Percent more than 100	percent=150	JSON error: percent invalid	Valid

22	/auto-cluster	TC-22	End-to-end clustering pipeline	algorithm="kmeans", remove-incomplete-rows=true	All files from step-1 to step-8 correctly saved, complete result submitted	Success
23	/auto-cluster	TC-23	Unknown algorithm	algorithm="xyz"	JSON error "unsupported algorithm"	Valid

4.1.3 Case Study Testing

To evaluate the practical performance of AutoClusterAPI in a real-world scenario, this study applied the system to the Customer Segmentation Dataset from Kaggle, curated by Vetri (vetrirah, 2025). The dataset contains demographic and behavioral attributes commonly used in marketing analytics, making it suitable for clustering-based customer profiling. A 20% random sample of complete records was extracted, yielding a working dataset of 1,333 rows. Sampling was used rather than the full dataset because several algorithms — particularly Spectral Clustering — exhibit high computational complexity that scales non-linearly with dataset size. Spectral Clustering requires constructing large similarity matrices and performing eigen-decomposition on the graph Laplacian, resulting in high memory consumption and time complexity that can reach quadratic to cubic growth (Pourkamali-Anaraki, 2020; Wu *et al.*, 2018; Khoa & Chawla, 2011; Li *et al.*, 2016). Limiting the dataset size allowed all eight algorithms to run under consistent experimental conditions.

Table 2. Sample entries from the Customer Segmentation Dataset

ID	Gender	Married	Age	Graduated	Profession	Experience	Spending Score	Family Size	Var1	Segment
461197	Male	No	28	No	Marketing	0.0	Low	1.0	Cat_3	D
463554	Female	No	53	No	Artist	7.0	Low	1.0	Cat_6	C
465488	Male	No	18	No	Healthcare	0.0	Low	4.0	Cat_6	D
461073	Female	No	33	No	Entertainment	1.0	Low	1.0	Cat_3	A
463196	Female	Yes	67	Yes	Engineer	1.0	Low	1.0	Cat_6	A

Each record represents an individual customer containing information such as gender, marital status, age, professional background, work experience, spending score, and family size. The clustering pipeline used a combination of numerical and categorical features selected for their analytical relevance. Numerical attributes — age, work_experience, and family_size — were chosen because they allow quantitative differentiation among customer groups. Categorical attributes — gender, ever_married, graduated, profession, and spending_score — were included for their relevance in describing demographic and socioeconomic characteristics used in segmentation practice. Several columns were deliberately excluded: id functions purely as an identifier and carries no discriminatory value for unsupervised tasks, while var_1 lacks clear semantic definition in the dataset documentation. The original segmentation label was also removed, as the experiment focuses exclusively on unsupervised clustering — AutoClusterAPI must identify inherent structure in the data without relying on preexisting category labels. After feature selection, the dataset was processed through the complete AutoClusterAPI pipeline, which includes column normalization, one-hot encoding, standardization, and PCA-based dimensionality reduction for visualization. All eight algorithms — K-Means, GMM, DBSCAN, OPTICS, Mean Shift, Spectral Clustering, Agglomerative Clustering, and BIRCH — were executed sequentially on the same preprocessed dataset. Each algorithm produced its own cluster assignments, subsequently visualized in two-dimensional space through PCA. The resulting scatter plots, presented in Figures 7 and 8, reveal distinct segmentation patterns generated by each method, highlighting differences in cluster shapes, densities, distributions, and separability.

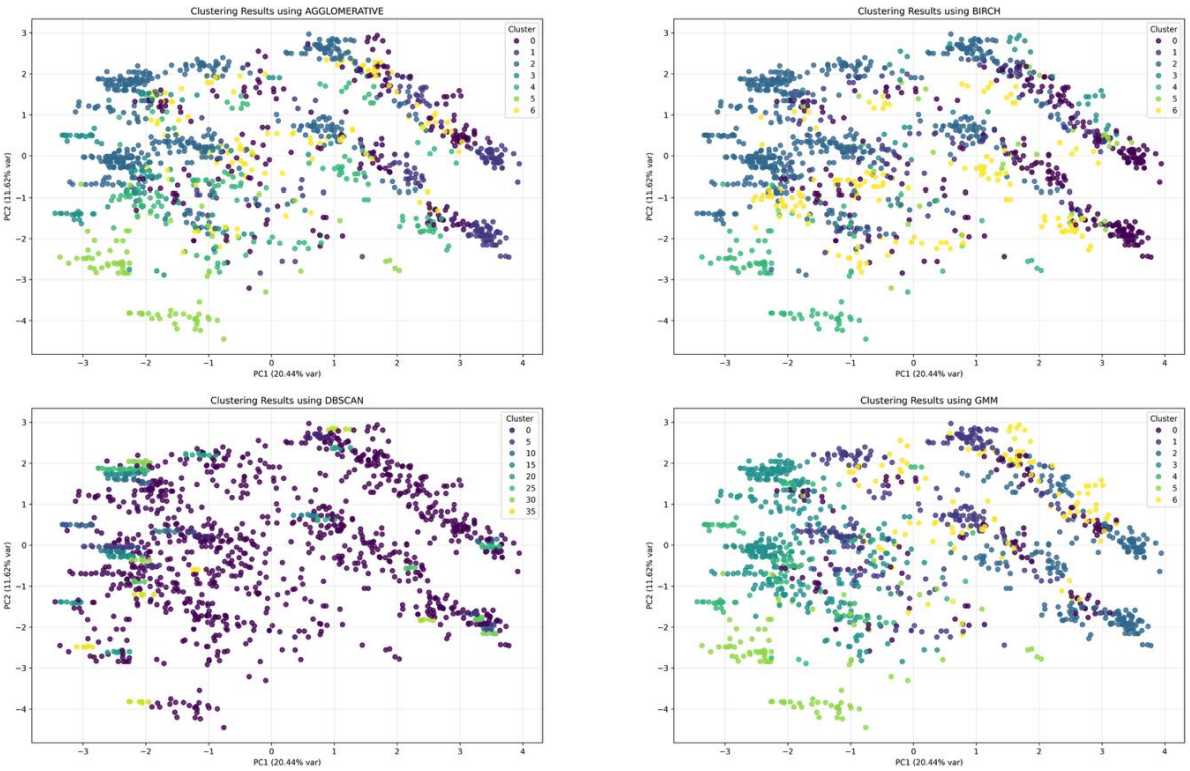

Figure 7. Clustering result using Agglomerative, BIRCH, DBSCAN and GMM

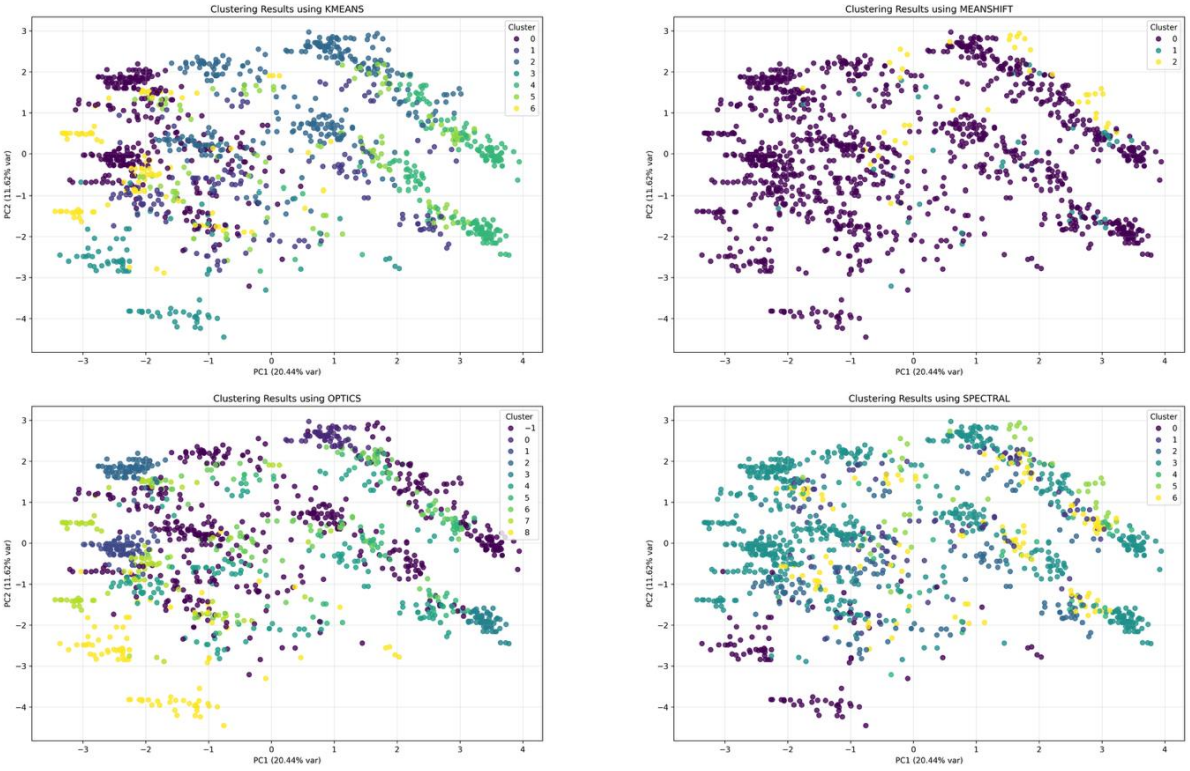

Figure 8. Clustering result using K-Means, MeanShift, OPTICS and Spectral.

4.2 Discussion

The test results show a consistent pattern across all 23 test cases. Under valid inputs, every endpoint produced the expected artifacts and returned correctly structured responses, confirming functional correctness across the entire pipeline. Error cases — unsupported algorithms, invalid column names, out-of-bounds percentages, and missing prerequisite files — were handled without unintended side effects, verifying that input validation is working as designed. Parallel session execution produced no interference between users, confirming that the stateless architecture holds under concurrent load. The /auto-cluster endpoint completed the full pipeline in a single call, demonstrating end-to-end automation without manual intervention at any

stage. The case study results further demonstrate that AutoClusterAPI can generate meaningful cluster structures from real-world data with minimal user effort. The eight algorithms produced visibly distinct segmentation patterns on the same customer dataset, as shown in Figures 7 and 8. Partition-based methods such as K-Means and GMM produced compact, well-separated clusters suitable for straightforward customer profiling. Density-based methods — DBSCAN and OPTICS — identified irregular cluster shapes and flagged outlier records that would otherwise be silently absorbed into the nearest cluster by distance-based algorithms. Mean Shift produced a variable number of clusters without requiring the user to predefine that number, which is useful in exploratory settings where the expected number of segments is unknown. Spectral Clustering captured non-linear boundaries that the other methods missed, though at a noticeably higher computational cost even on the reduced 1,333-row sample. Agglomerative Clustering and BIRCH both produced stable results, with BIRCH showing particular efficiency on this dataset size.

These differences across algorithms are not merely academic. In practice, the choice of algorithm directly affects which customer segments are identified and how they are interpreted. AutoClusterAPI makes this comparison straightforward: users can run all eight algorithms sequentially through the same pipeline and compare their PCA scatter plots and cluster profiles side by side, without rewriting any code between runs. This is a practical advantage over conventional programming-based workflows, where switching algorithms typically requires modifying scripts, re-running preprocessing steps, and manually reorganizing outputs. The system does not prescribe which algorithm is best — that judgment remains with the analyst — but it removes the technical friction that would otherwise prevent non-expert users from making that comparison at all. One limitation worth acknowledging is the computational constraint imposed by Spectral Clustering. Even at 1,333 rows, the algorithm required noticeably more time than the others. For datasets larger than a few thousand rows, users should either apply sampling — as done in this study — or consider switching to approximation-based variants documented in the literature (Pourkamali-Anaraki, 2020; Wu *et al.*, 2018). AutoClusterAPI does not currently enforce dataset size limits or provide automatic algorithm recommendations based on data scale, which represents a clear direction for future development.

5. Conclusion and Recommendations

The expectations outlined in the Introduction — namely the development of a clustering framework that is both accessible to non-technical users and sufficiently flexible for customized analytical workflows — have been fully realized through the implementation and evaluation of AutoClusterAPI. The system successfully provides an end-to-end clustering pipeline, accessible through intuitive RESTful endpoints, that automates the entire process from data ingestion and preprocessing to clustering, profiling, and visualization. These outcomes directly align with the initial motivation to bridge ease-of-use and customizability while reducing the technical barriers commonly associated with clustering tasks. The black-box functional testing demonstrates that all endpoints behave as intended, validating the robustness, correctness, and reliability of the system under both valid and invalid inputs. The case study using a real-world customer segmentation dataset further substantiates the system's practical applicability: AutoClusterAPI executed all eight clustering algorithms, generated interpretable cluster profiles, and produced PCA-based scatter plots that visually distinguish segmentation patterns across methods. This coherence between the goals articulated in the Introduction and the empirical outcomes confirms that AutoClusterAPI effectively fulfills its intended role.

Beyond its core objectives, this research opens several avenues for future development. Potential enhancements include extending support for additional clustering algorithms, incorporating automated hyperparameter tuning, enabling streaming data processing, and enriching the visual analytics component with interactive dashboards. Given its modular and stateless architecture, AutoClusterAPI can also be adapted for deployment in distributed or cloud-based environments, integrated into business intelligence systems, or expanded into a comprehensive clustering microservice for large-scale enterprise applications. These prospects indicate that AutoClusterAPI not only addresses current practical needs but also provides a scalable foundation for further research and development in operational machine learning systems.

Acknowledgment

The authors would like to express their sincere gratitude to Politeknik Negeri Malang for providing financial support for this research through the institutional research grant funded under SP DIPA-023.18.2.677606/2024, administered by the Center of Research and Community Service.

References

Aggarwal, C. C. (2016). An introduction to recommender systems. In *Recommender systems: The textbook* (pp. 1-28). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-29659-3_1

Agrawal, K. P., Garg, S., Sharma, S., & Patel, P. (2016). Development and validation of OPTICS based spatio-temporal clustering technique. *Information Sciences*, *369*, 388-401. https://doi.org/10.1016/j.ins.2016.06.048.

Ahmed, M., Seraj, R., & Islam, S. M. S. (2020). The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, *9*(8), 1295. https://doi.org/10.3390/electronics9081295

Alla, M. (2025). Designing High-Throughput FastAPI Gateways for Microservice Communication. *Journal of Computer Science and Technology Studies*, *7*(7), 823-828. https://doi.org/10.32996/jcsts.2025.7.7.88.

Amershi, S., Cakmak, M., Knox, W. B., & Kulesza, T. (2014). Power to the people: The role of humans in interactive machine learning. *AI magazine*, *35*(4), 105-120.

Aravind, H., Rajgopal, C., & Soman, K. (2010). A simple approach to clustering in excel. *International Journal of Computer Applications*, *11*(7), 19–25.

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, *41*(3), 1-58. https://doi.org/10.1145/1541880.1541882.

Demšar, J., Curk, T., Erjavec, A., Gorup, Č., Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., & Zupan, B. (2013). Orange: Data mining toolbox in Python. *Journal of Machine Learning Research*, *14*(1), 2349–2353.

Deng, D. (2020, September). DBSCAN clustering algorithm based on density. In *2020 7th international forum on electrical engineering and automation (IFEEA)* (pp. 949-953). IEEE. https://doi.org/10.1109/IFEEA51475.2020.00199.

Few, S. (2009). *Now you see it: Simple visualization techniques for quantitative analysis*. Analytics Press.

Franciska, I., & Swaminathan, B. (2017, May). Churn prediction analysis using various clustering algorithms in KNIME analytics platform. In *2017 Third International Conference on Sensing, Signal Processing and Security (ICSSS)* (pp. 166-170). IEEE. https://doi.org/10.1109/SSPS.2017.8071585.

Garg, S., Ahuja, R., Singh, R., & Perl, I. (2024). An effective deep learning architecture leveraging BIRCH clustering for resource usage prediction of heterogeneous machines in cloud data center. *Cluster Computing*, *27*(5), 5699-5719. https://doi.org/10.1007/s10586-023-04258-6.

Géron, A. (2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media.

Greenacre, M., Groenen, P. J., Hastie, T., d'Enza, A. I., Markos, A., & Tuzhilina, E. (2022). Principal component analysis. *Nature Reviews Methods Primers*, *2*(1), 100. https://doi.org/10.1038/s43586-022-00184-w.

Hajihosseinlou, M., Maghsoudi, A., & Ghezelbash, R. (2024). Intelligent mapping of geochemical anomalies: Adaptation of DBSCAN and mean-shift clustering approaches. *Journal of Geochemical Exploration*, *258*, 107393. https://doi.org/10.1016/j.gexplo.2024.107393.

Kablan, M., Caldwell, B., Han, R., Jamjoom, H., & Keller, E. (2015). Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (pp. 49–54). ACM. https://doi.org/10.1145/2785989.2785993

Khoa, N. L. D., & Chawla, S. (2011). *Large scale spectral clustering using approximate commute time embedding*. arXiv preprint arXiv:1111.4541.

Li, Y., Huang, J., & Liu, W. (2016, February). Scalable sequential spectral clustering. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 30, No. 1). https://doi.org/10.1609/aaai.v30i1.10298.

Marchev, V. (2021). An empirical exploration of data segmentation and clustering methods: Insights from SPSS Modeler. *Vanguard Scientific Instruments in Management*, *17*(1), 1314–0582.

Martin-Lopez, A., Segura, S., & Ruiz-Cortés, A. (2021, July). RESTest: automated black-box testing of RESTful web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 682-685). https://doi.org/10.1145/3460319.3469082.

Peralta, J. H. (2023). *Microservice APIs: Using Python, Flask, FastAPI, OpenAPI and more*. Simon and Schuster.

Pourkamali-Anaraki, F. (2020). Scalable spectral clustering with Nyström approximation: Practical and theoretical aspects. *IEEE Open Journal of Signal Processing*, *1*, 242-256. https://doi.org/10.1109/OJSP.2020.3039330.

Raya-Tapia, A. Y., López-Flores, F. J., Ramírez-Márquez, C., & Ponce-Ortega, J. M. (2025). Programming for Clustering: Python, R, and MATLAB. In *Machine Learning and Clustering for a Sustainable Future: Applications in Engineering and Environmental Science* (pp. 51-99). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-032-03876-0_3.

Tang, C., Li, Z., Wang, J., Liu, X., Zhang, W., & Zhu, E. (2022). Unified one-step multi-view spectral clustering. *IEEE Transactions on Knowledge and Data Engineering*, *35*(6), 6449-6460. https://doi.org/10.1109/TKDE.2022.3172687.

Tokuda, E. K., Comin, C. H., & Costa, L. D. F. (2022). Revisiting agglomerative clustering. *Physica A: Statistical mechanics and its applications*, *585*, 126433. https://doi.org/10.1016/j.physa.2021.126433.

Tragura, S. J. C. (2022). *Building Python microservices with FastAPI: Build secure, scalable, and structured Python microservices from design concepts to infrastructure*. Packt Publishing.

vetrirah. (2025). *Customer — Customer segmentation dataset* [Dataset]. Kaggle. https://www.kaggle.com/datasets/vetrirah/customer

Wu, L., Chen, P. Y., Yen, I. E. H., Xu, F., Xia, Y., & Aggarwal, C. (2018, July). Scalable spectral clustering using random binning features. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 2506-2515). https://doi.org/10.1145/3219819.3220090.

Xu, D., & Tian, Y. (2015). A comprehensive survey of clustering algorithms. *Annals of data science*, *2*(2), 165-193. https://doi.org/10.1007/s40745-015-0040-1.

Yang, M. S., Lai, C. Y., & Lin, C. Y. (2012). A robust EM clustering algorithm for Gaussian mixture models. *Pattern recognition*, *45*(11), 3950-3961. https://doi.org/10.1016/j.patcog.2012.04.031.

Yuan, B. (2025, May). Enhanced Supply Chain Risk Management Using K-Means Clustering and Tableau: A Hybrid Framework for Big Data Visualization. In *2025 2nd International Conference on Intelligent Computing and Robotics (ICICR)* (pp. 509-513). IEEE. https://doi.org/10.1109/ICICR65456.2025.00094.