HOCHSCHULE EMDEN·LEER

Data Science and Analytics

Sentiment Analysis

Author 1 : Bony Martin
Matriculation Number : 7026527
Author 2 : Bharathraj Govindaraj
Matriculation Number : 7026834
Course of Studies: Industrial Informatics

Examiners: Prof. Dr. Elmar Wings
Prof. Dr. Armando Walter Colombo
Submission date: January 8, 2026

University of Applied Sciences Emden/Leer · Faculty of Technology · Electrical Engineering and Computer Science Department
Constantiaplatz 4 · 26723 Emden · http://www.hs-emden-leer.de

Abstract

Customer reviews on platforms such as Google Reviews and delivery apps contain valuable signals about food quality, service performance, and managerial decisions. In practice, these signals are often processed manually, which is slow and inconsistent. This report documents local-first analytics pipeline that transforms CSV exports of review text into actionable, evidence-based insights for restaurant owners and managers. The pipeline performs robust preprocessing (text cleaning, minimum length filtering, deduplication, optional language filtering, date and star parsing), computes baseline sentiment with VADER, and calls a locally hosted large language model (LLM) via an Ollama-compatible endpoint to extract short evidence quotes for *Kitchen*, *Service*, and *Management*. To reduce disagreement between lexical sentiment and LLM labels, the implementation combines both signals using a gated fusion strategy and stores intermediate progress in a cache file for reproducibility and resume capability. Each run produces a timestamped output folder with a processed dataset, owner summary (JSON and readable text), charts, and an optional weekly email report with deduplicated attachments. The system is positioned in the RAMI 4.0 Functional Layer as a decision-support analytics capability. The report explains design choices, data and information flow, evaluation approach, and practical limitations, and provides a complete deliverable package for submission.

Contents

List of Figures

Listings

Acronyms

CSV Comma-Separated Values

KDD Knowledge Discovery in Databases

LLM Large Language Model

NLP Natural Language Processing

RAMI Reference Architectural Model Industrie 4.0

RRI Restaurant Review Insights

SMTP Simple Mail Transfer Protocol

VADER Valence Aware Dictionary and sEntiment Reasoner

JSON JavaScript Object Notation

TXT TeXT File

API Application Programming Interface

HTTP HyperText Transfer Protocol

TF-IDF Term Frequency-Inverse Document Frequency

SVM Support Vector Machine

BERT Bidirectional Encoder Representations from Transformers

NLTK Natural Language Toolkit

Chapter 1

Introduction

1.1 Use case and motivation

This project is part of the practice series of Pizzahaus , a company owned and operated by Mr. Thomas Porrmann.Sentiment analysis provides substantial practical value by transforming large volumes of unstructured textual data into actionable operational insights. In the context of the client's use case, restaurants receive continuous customer reviews via online platforms, which are typically stored as raw Comma-Separated Values (CSV) exports and remain difficult to evaluate manually over time. Building on established sentiment analysis practices used in industrial and service-oriented environments, this project applies Natural Language Processing (NLP) and text mining techniques to the restaurant domain. The proposed sentiment analysis system automatically analyzes customer reviews to identify sentiment trends, recurring issues, and indicators of customer satisfaction. Feedback is systematically categorized into three operational domains—Kitchen, Service, and Management to ensure relevance for daily restaurant operations. By detecting patterns such as delivery delays, food quality perceptions, or service friendliness, the system supports early identification of improvement opportunities. In addition to sentiment scoring, the proposed solution generates evidence-based insights and visual summaries tailored to restaurant owners. Automated weekly reports distributed via an Simple Mail Transfer Protocol (SMTP)-configurable email service further enhance accessibility of results. Overall, the system aligns with the client's use case by enabling data-driven decision-making and continuous quality improvement through scalable sentiment analysis.

1.2 Challenges

The practical implementation of this project revealed several challenges and limitations related to data quality, scalability, and model reliability. Customer review data is inherently unstructured, containing mixed languages, typographical errors, duplicates, noise, and highly variable text lengths, which complicates preprocessing and analysis. Additionally, sentiment ambiguity was frequently observed, such as neutral or factual review texts accompanied by strongly positive or negative ratings, which makes a complex sentiment interpretation . From a system perspective, the large volume of review data posed performance constraints during ingestion, as even systems with adequate hardware specifications experienced slowdowns when processing inputs through the language model pipeline.

Language diversity in the dataset further introduced limitations, particularly when applying classical sentiment analysis methods such as VADER, which primarily support English and, therefore, excluded non-English reviews. Furthermore, passing large amounts of textual data to the Large Language Model (LLM) increased memory usage and led to hallucinated or irrelevant outputs, potentially reducing the trustworthiness of generated insights. To mitigate this issue, a strict operational mode was enforced, constraining the LLM to predefined features and outputs relevant to the project objectives. These challenges highlight the importance of robust preprocessing, language handling, and controlled model behavior when generating reliable evidence-based insights from customer reviews.

1.3 Results produced by the system

In one run, the software generates a per-run output folder containing: a processed review table, an owner summary (JSON + readable TXT), three charts (stars, sentiment distribution, monthly sentiment trend), and a weekly email text with attachments.

1.4 Deliverables produced

The delivered project package contains:

- **Report:** `report/main.tex`, `report/references.bib`, and the compiled PDF.
- **Source code:** the complete Python pipeline and helper scripts in `src/`.
- **Configuration template:** `.env.example` to document required environment variables.
- **Figures:** the architecture diagram and the end-to-end pipeline flowchart used in the report.
- **SBOM:** `requirements.txt` listing Python dependencies.
- **Documentation:** `README.md` and `README.txt` describing the repository and how to run it.

Part I

Domain Knowledge - Application

Chapter 2

Application

2.1 Knowledge domain

The application domain of this project lies within the **restaurant operations** sector, with a specific focus on the analysis of customer feedback to support operational decision-making. The primary stakeholders include restaurant owners and management, who utilize the derived insights to inform strategic decisions related to pricing, staffing, and overall quality improvements. In addition, kitchen and service teams benefit from actionable, evidence-based feedback that highlights strengths and areas for improvement in food quality, service efficiency, and customer experience. By addressing the needs of both managerial and operational stakeholders, the system supports continuous performance monitoring and data-driven optimization within restaurant environments.

2.2 State of the art

Existing approaches to customer review analysis in the hospitality sector typically include built-in platform dashboards, manual inspection of individual reviews, and cloud-based sentiment analytics services. While platform dashboards provide high-level metrics, they often lack transparency and domain-specific interpretability, whereas manual review reading is time-consuming and does not scale with large volumes of data. Cloud-based sentiment analytics solutions offer advanced processing capabilities but raise concerns related to data privacy, cost, and limited customization. In contrast, the proposed system is positioned as a **privacy-friendly, locally deployed analytics pipeline** tailored to restaurant operations. It combines classical sentiment analysis with modern language models to balance interpretability and expressiveness. VADER sentiment scoring [4] is employed to generate fast, rule-based baseline sentiment scores that are transparent and suitable for large-scale processing. Complementing this, a locally hosted LLM accessed via an Ollama-compatible API is used to extract evidence-based quotations and domain-specific insights across Kitchen, Service, and Management categories. The local deployment ensures that sensitive customer data remains on-premise, addressing privacy and compliance requirements. Furthermore, this hybrid design allows restaurant owners to obtain trustworthy summaries and actionable insights without reliance on external cloud services, positioning as a scalable and cost-effective alternative to existing solutions.

2.3 What the analytics is doing

The proposed system supports **management and operational decision-making** by systematically consolidating customer feedback into domain-specific themes and actionable weekly insights. It functions as a diagnostic tool by identifying recurring issues such as long waiting times, price–value mismatches, or service-related complaints across large volumes of reviews. In addition, the sentiment analysis system provides optimization support by prioritizing improvement areas based on sentiment trends and evidence extracted directly from customer feedback. This enables managers to focus on high-impact actions supported by concrete examples rather than anecdotal observations. Furthermore, the system facilitates continuous monitoring by tracking sentiment indicators over time, allowing stakeholders to assess the effectiveness of implemented changes. Through this combination of diagnosis, optimization, and monitoring,The proposed system transforms unstructured review data into structured intelligence that supports data-driven restaurant management.

2.4 Positioning within reference architectures

In the project documentation,The sentiment analysis system is positioned within Reference Architectural Model Industrie 4.0 (RAMI) as a **Functional Layer** capability for customer feedback analytics and decision support. The placement in the Functional Layer is justified by the role of the developed system that processes information and provides decision-support services: (i) it implements the analytics logic itself (preprocessing, sentiment scoring, LLM-based evidence extraction, fusion, summarization, and chart generation), (ii) it exposes a consistent set of outputs (processed dataset, owner summary artifacts, and report email) that can be consumed by humans or downstream tools, and (iii) it remains independent from physical assets and field-level integration, since it operates on exported review data rather than directly controlling or representing an asset in the Asset/Integration layers. Therefore, the developed solution is best described as a functional analytics component with clear inputs and outputs, providing business value through decision support.

2.5 System architecture (RAMI 4.0 view)

Figure 2.1 presents a simplified, RAMI 4.0-inspired mapping of the end-to-end analytics workflow. In RAMI 4.0 terms, the **core product capability** of this project—turning raw customer reviews into structured, actionable insights—is positioned in the **Functional Layer**. The surrounding boxes in the figure illustrate the adjacent layers that the product *interfaces with*: the **Information Layer** provides the required input data and configuration (e.g., `.env` settings and review CSV files), while the **Business Layer** represents how the generated insights are used to support owner/manager decisions (e.g., improvement actions derived from evidence).

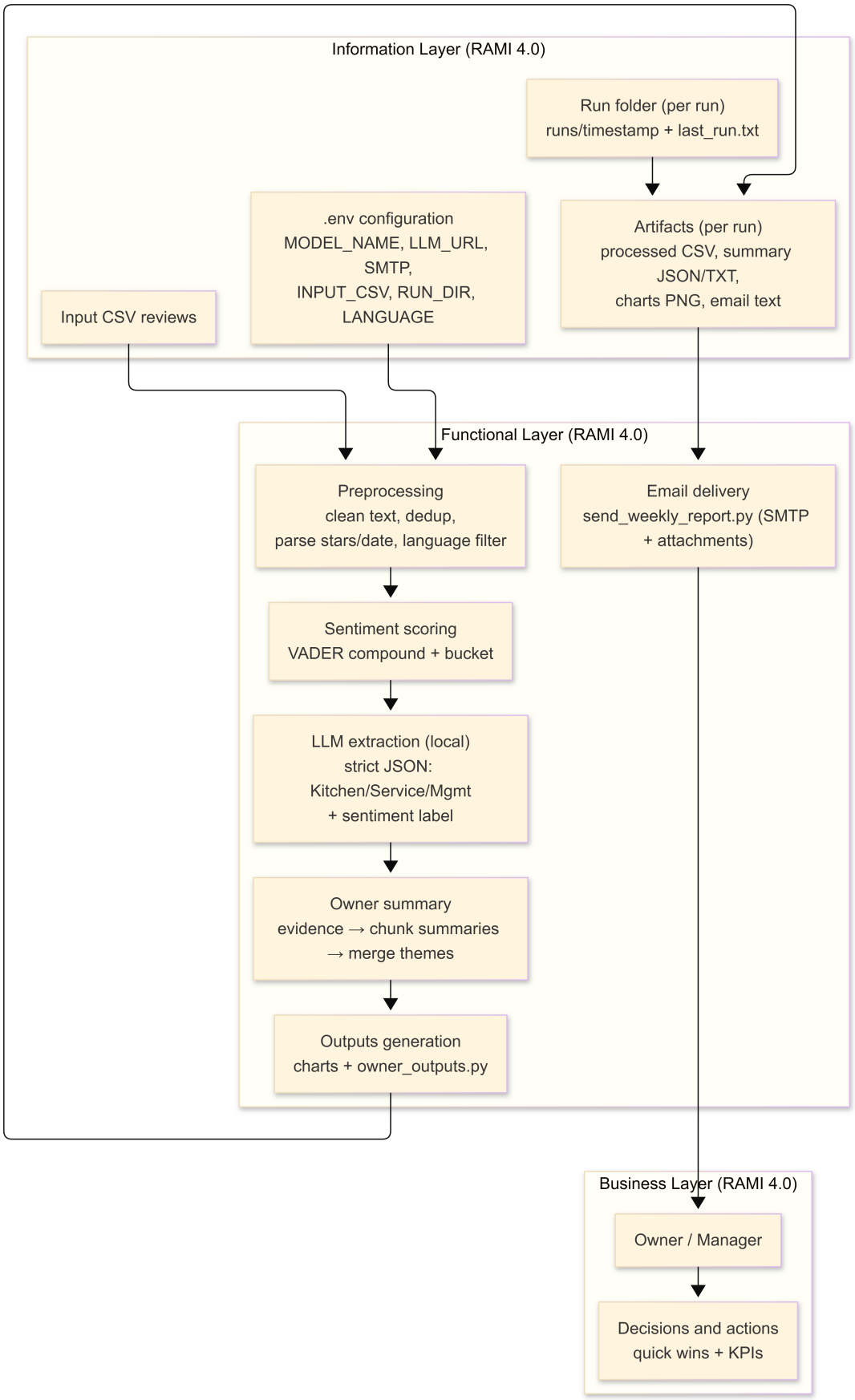

Figure 2.1: Simplified RAMI 4.0-inspired architecture mapping: Information inputs → Functional analytics (core product) → Business usage (decision support).

The above architecture is intentionally simplified to highlight the relevant RAMI layers for this project; the implemented software module is located in the Functional Layer, while the Information and Business layers indicate the upstream inputs and downstream usage context.

2.6 Infrastructure integration

2.6.1 Data source and execution environment

This project does not require dedicated physical hardware sensors. The "data acquisition" is performed by importing CSV exports of customer reviews. The execution environment is a standard workstation/laptop running Python and (optionally) a local LLM runtime. In this sense, the system is designed as a lightweight analytics component that can be executed on commonly available computing devices, without special infrastructure requirements.

2.6.2 Data source (review exports)

The primary data source for the system consists of customer review exports obtained from online review platforms, typically provided in CSV format. At a minimum, the dataset is expected to contain a `review_text` column, with optional metadata such as numerical ratings (`stars`) and timestamps (`date`). These additional fields enable richer reporting, such as star-distribution summaries and time-based trend analyses. The pipeline also supports flexible input handling by selecting either an explicitly configured CSV file or, if not configured, automatically detecting the newest available review export in the project directory.

2.6.3 Execution environment

The pipeline is implemented in Python and can be executed in a standard local environment, such as a Windows or Linux workstation. Dependencies are managed via a `requirements.txt` file to support reproducible setup. Outputs are written into a dedicated per-run folder (`RUN_DIR`) so that repeated runs remain isolated and comparable, and intermediate cache files allow the pipeline to resume after interruptions without reprocessing already completed reviews.

2.6.4 Local LLM endpoint

The LLM is accessed via an Ollama-compatible endpoint (HTTP POST `/api/generate`). The software requests strict JSON outputs to reduce parsing ambiguity and to ensure that extracted fields can be consumed deterministically by downstream processing steps. In the implemented workflow, the LLM is used to extract domain-oriented evidence (e.g., Kitchen, Service, and Management) and to provide an overall sentiment label, which complements lexicon-based sentiment scoring and supports owner-focused summaries.

2.6.5 Major infrastructure requirements for the analytics

For reliable operation in a larger infrastructure, the analytics component must satisfy structural, behavioural, functional, and technological requirements to ensure repeatable execution, scalability, and stakeholder trust.

Structural requirements. Configuration must be separated from code: all runtime parameters (input selection, language mode, LLM endpoint, SMTP) must be externalized in `.env` for portability. The run-based folder structure must be preserved for auditability and to prevent output collisions. Interfaces must remain stable through standardized formats (CSV for tabular outputs, JSON for structured summaries) to support other scripts and external tools.

Behavioural requirements. The pipeline must be deterministic and resilient to interruptions: resume-safe caching, consistent handling of missing/malformed inputs, and controlled retry logic for LLM calls. Logging is required at each major step (input detection, preprocessing, extraction, summarization, chart generation, reporting) for monitoring and troubleshooting. Duplicate work must be avoided by skipping already processed reviews when resuming.

Functional requirements. The system must (i) ingest review data, (ii) clean/normalize text, (iii) compute baseline sentiment (VADER), (iv) extract domain evidence (kitchen, service, management) using an LLM with strict JSON constraints, (v) aggregate into owner-ready summaries, and (vi) generate visualizations and reporting artifacts. Explainability is required: insights must remain traceable to review evidence so users can verify conclusions.

Technological requirements. A Python runtime with required dependencies (Pandas, NLTK, Matplotlib, Requests) is needed. A local LLM runtime must be reachable via HTTP for AI processing. Outbound SMTP connectivity must be available for automated reporting. Adequate compute is required for repeated LLM calls; model size and inference speed must match latency/hardware/cost constraints, hence the use of a smaller local model for practical weekly reporting.

Part II

Domain Knowledge - Machine Learning/Algorithms

Chapter 3

Algorithms and method selection

3.1 Alternative approaches (overview)

Several method families can be used for sentiment analysis on review text, each with different trade-offs in accuracy, interpretability, and operational effort. **Lexicon-based sentiment** (e.g., VADER) is fast, interpretable, and requires no training data, but it is limited for complex context or domain-specific phrasing. [4] **Classical supervised ML** (e.g., TF–IDF + SVM/logistic regression) can perform well with labeled datasets, but it requires training data, feature engineering, and periodic re-training; evidence extraction is not native. **Transformer classifiers** (e.g., BERT-based models) provide strong contextual understanding, but they have higher compute requirements and are still primarily label-focused unless combined with additional explainability methods. [2] **LLM-based analysis** can provide both sentiment labels and structured evidence, but it must be constrained to prevent hallucinations and to ensure deterministic parsing. [6]

3.2 Why hybrid approach was chosen

This project uses a hybrid design combining VADER and a locally hosted LLM. The selection is motivated by practical requirements of owner-oriented reporting. **Efficiency and stability:** VADER provides a lightweight baseline signal that is fast to compute and easy to interpret. [4] **Actionable evidence:** the LLM is used to extract short pros/cons evidence grouped into Kitchen, Service, and Management, enabling transparent insights beyond a single sentiment label. **Local execution:** running the LLM locally (Ollama-compatible endpoint) reduces dependency on external cloud services and supports data control. **Deterministic outputs:** strict JSON constraints reduce ambiguity and allow the pipeline to parse and store extracted evidence reliably.

3.3 Baseline sentiment (VADER)

VADER returns a *compound* score $v \in [-1, 1]$ and is mapped to sentiment buckets using common thresholds: Positive ($v \geq 0.05$), Negative ($v \leq -0.05$), otherwise Neutral. [4]

3.4 Structured evidence extraction (LLM)

The LLM is prompted to extract **short, direct evidence quotes** from the review text, or return `n/a` when no relevant information exists. Evidence is structured into three operational domains: Kitchen, Service, and Management. Prompts explicitly restrict the model to the provided review content and instruct it to avoid inventing facts or assumptions. To support automated processing, responses are required in strict JSON format.

3.5 Fusion of VADER and LLM sentiment

To reduce disagreement between lexical sentiment scores and LLM predictions, the pipeline employs a **gated fusion** strategy. If VADER is "unsure" (i.e., $|v| < 0.25$), the fusion gives more weight to the LLM output (0.25 VADER, 0.75 LLM). Otherwise, when VADER shows a stronger sentiment signal, the fusion slightly favors VADER (0.55 VADER, 0.45 LLM). The fused score is computed as a weighted sum and then mapped to a final fused label for reporting.

Part III

Domain Knowledge - Tools

Chapter 4

Tools and Libraries

The product is implemented in Python and combines lightweight data processing, interpretable sentiment scoring, controlled LLM extraction, reporting, and optional email delivery. For structured data handling, **pandas** is used to load review CSV files, clean and filter rows, deduplicate reviews, parse dates, and write outputs such as cached progress files and `reviews_processed.csv` [5], [7]. For baseline NLP, **NLTK** provides the VADER sentiment analyzer and stopword lists [1], enabling a transparent sentiment score (compound) and simple language filtering.

Visual outputs are generated with **matplotlib**, producing charts for star distribution, sentiment distribution, and monthly sentiment trends. The local LLM integration is implemented with **requests**, calling an Ollama-compatible HTTP endpoint to extract evidence (kitchen/service/management) in strict JSON and to generate an owner summary. Product robustness is supported by Python standard modules such as `pathlib`/`os` (paths and run folders), `json`/`re` (safe parsing and validation), `datetime`/`time` (timestamps and pacing), and `subprocess` (running helper scripts). For operational delivery, the reporting feature uses `smtplib` and `EmailMessage` to send an email with deduplicated attachments, configured via `.env`.

Part IV

Methodology and Development

Chapter 5

KDD process mapping

The Knowledge Discovery in Databases (KDD) framing is applied to the review analytics pipeline to describe how raw customer feedback is transformed into actionable outputs. The overall steps follow the classical KDD stages: selecting the data source, preparing and cleaning the data, transforming it into analysis-ready form, performing the core analytics, and validating the produced results. [3]

5.1 KDD mapping overview

Figure 5.1 summarizes how the implemented pipeline aligns with the KDD stages.

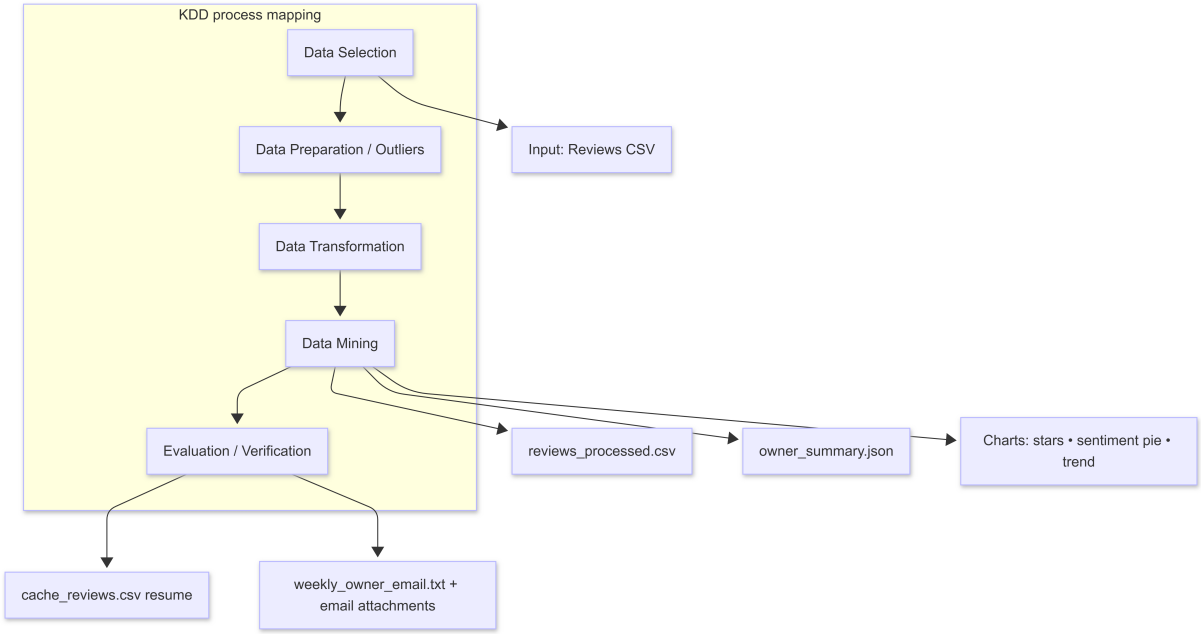

Figure 5.1: KDD mapping of the review analytics pipeline.

5.2 Data selection and database representation

The pipeline takes an input CSV file containing customer review text and optional metadata such as star ratings and timestamps. The input selection is deterministic: if `INPUT_CSV` is provided via the `.env` configuration file, that file is used; otherwise, the pipeline automatically selects the newest `*.csv` file in the project folder while excluding known output files (e.g., cache and processed CSVs). This approach supports both controlled experiments (fixed input) and practical operation (latest export) without changing the code.Data selection and database representation

5.2.1 Input data

The primary input to the pipeline is a CSV export containing customer review text and, when available, metadata such as star ratings and timestamps. In the implementation, the review text column is treated as mandatory because it is required for both VADER scoring and LLM-based evidence extraction. Ratings and timestamps are optional; however, when present they enable additional outputs such as star-distribution charts and time-based trend analysis. The main script supports two selection modes in order to cover both reproducible experiments and practical day-to-day usage:

- **Explicit input:** via the environment variable `INPUT_CSV`. This mode is used when a specific dataset must be processed (e.g., for controlled evaluation, debugging, or reporting).
- **Auto-detection:** the newest `*.csv` file in the project directory is selected (excluding known output names). This mode supports quick operation when new exports are placed in the project folder, without requiring code changes.

In both modes, the selected CSV is loaded using `pandas`, and the pipeline validates the required columns. If the input uses different column names for review text, rating, or date, the implementation applies a column-detection and renaming step so that downstream processing consistently operates on the expected internal column names.

5.2.2 Data representation

The pipeline preserves the original review rows and extends the dataset with derived attributes that are produced during preprocessing, sentiment scoring, and extraction. This enriched representation is written to a processed output CSV and is designed to support both quantitative reporting (scores, distributions, trends) and qualitative reporting (evidence quotes that justify the summary). The derived attributes include:

- **Cleaned review text:** a normalized version of the raw review text (e.g., whitespace normalization and removal of invalid placeholders) that is used consistently for scoring and extraction.
- **Parsed date and star ratings:** timestamps are parsed from mixed formats into a consistent date representation for aggregation, and star ratings are normalized to integer values on the 1–5 scale.
- **VADER sentiment signal:** the VADER compound score and a mapped bucket label (Positive/Neutral/Negative) based on defined thresholds.

- **LLM evidence fields per domain:** structured pros/cons evidence extracted for Kitchen, Service, and Management, alongside an overall sentiment label and confidence returned by the LLM using strict JSON outputs.
- **Fused score and label:** a combined sentiment result that integrates the lexicon-based VADER signal with the LLM-derived label/confidence, producing a final fused score and fused label for reporting.

In addition to the processed dataset, the system generates owner-oriented outputs where the extracted evidence is consolidated into themes and transformed into a structured owner summary (`owner_summary.json`) as well as a human-readable text version.

5.2.3 Outputs and storage

All outputs are written into a dedicated per-run folder to keep executions isolated and comparable. In the implementation, each pipeline execution creates a timestamped run directory under `runs/`, and the path of the most recent run is recorded in `runs/last_run.txt`. This approach ensures that helper scripts (such as email delivery) can reliably locate the latest artifacts without manual configuration. The run folder acts as a lightweight "experiment record" containing(reviews processed.csv, owner summary.json, owner summary readable.txt, three chart PNG files, and weekly email text. To support robustness and resume behavior on larger datasets, the pipeline also maintains a cache file (`cache_reviews.csv`) that stores intermediate processing results. This structure improves traceability (it is clear which run produced which artifacts), supports reproducibility by preserving run outputs, and avoids accidental overwriting when the pipeline is executed repeatedly during development or routine reporting.

5.3 Data preparation

Before analytics, the pipeline performs data preparation to improve robustness and to reduce noise. Review text is cleaned by normalizing whitespace and removing empty or very short reviews according to a minimum text length threshold. Duplicate reviews are removed to avoid bias in sentiment distribution and downstream summaries. When enabled, a lightweight language filter (auto/de/en) keeps the review set consistent for the selected analysis language. Star values are validated and clamped to the supported scale of 1–5 stars, ensuring that invalid entries do not distort star distribution statistics or trend plots.

5.4 Data transformation

For trend analysis and reproducibility, the pipeline normalizes key fields into consistent internal representations. Date values are parsed from mixed formats and converted into a uniform `date` field used for monthly aggregation and chart generation. In addition, review text is normalized prior to scoring and LLM-based extraction so that sentiment scoring and evidence extraction operate on comparable text input across the dataset.The data transformation step converts raw review exports into a consistent, analysis-ready format so that scoring, extraction, charts, and summaries behave reliably across different CSV sources.

5.4.1 Schema validation and column normalization

Before transforming values, the pipeline validates the incoming CSV schema and ensures that the required fields are available. Review exports from different platforms may use different column names (e.g., different names for review text, rating, or timestamp). Therefore, the pipeline performs a column-detection and renaming step to map the source columns into consistent internal field names. This normalization prevents errors in later stages and allows the same pipeline to run across multiple datasets without manual edits.

5.4.2 Review text normalization and filtering

The review text is transformed into a consistent representation for both VADER scoring and LLM-based extraction. Text normalization includes trimming leading and trailing whitespace, collapsing repeated whitespace, and removing invalid placeholders (for example, the literal string `nan` or empty strings). To reduce noise and avoid unstable scoring on extremely short inputs, the pipeline filters very short reviews using the configured threshold `MIN_TEXT_LEN`. These transformations ensure that sentiment scoring and evidence extraction operate on comparable, meaningful inputs across the dataset.

5.4.3 Duplicate handling

To prevent repeated entries from biasing distributions, trends, and the owner summary, the pipeline removes duplicate reviews. Deduplication is performed using the normalized review text, ensuring that small whitespace differences do not allow the same review to appear multiple times. This step improves the reliability of aggregated results, especially when exports contain repeated rows or when review sources are merged.

5.4.4 Date parsing and time normalization

For trend computations, the pipeline transforms the timestamp column into a normalized datetime representation. Review datasets often contain mixed date formats; the implementation applies robust parsing to handle these variations and then derives a consistent date field used for monthly aggregation and visualization. When dates are missing or cannot be parsed reliably, the pipeline still produces non-temporal outputs (processed dataset, sentiment distribution, and owner summary), while time-series charts are limited to reviews with valid timestamps. This design avoids failing the full run due to partial timestamp quality.

5.4.5 Star rating normalization

When star ratings are present, values are converted to numeric form and normalized to the valid 1–5 range by clamping out-of-range values. This transformation ensures consistent star distributions and prevents invalid inputs from distorting reporting. Star ratings are treated as metadata for visualization and descriptive statistics; they are not used as evidence sources for LLM extraction.

5.4.6 Language and label normalization

The pipeline supports optional language filtering (`LANGUAGE=de/en/auto`) so that scoring and extraction can be aligned to the chosen analysis language. In addition, outputs produced by the LLM are normalized before fusion and reporting. In practice, LLM responses may contain small variations in label spelling or language; therefore, the pipeline maps LLM-returned sentiment labels into a consistent internal vocabulary. This prevents aggregation errors and ensures that the fused label and downstream charts use stable categories across runs.

Overall, the data transformation stage ensures that the dataset entering the data mining loop is clean, standardized, and traceable. It reduces avoidable variability caused by export differences (schema, date format, duplicates) and stabilizes the input space for both lexicon-based scoring and LLM-based extraction.

5.5 Data mining

The core analytics step computes sentiment signals and structured evidence from each review. First, a VADER compound score is computed and mapped to a coarse sentiment bucket (Positive/Neutral/Negative) using standard VADER thresholds. [4] In parallel, the LLM produces a sentiment label with confidence and extracts short evidence quotes grouped by domain (Kitchen, Service, Management). The LLM label is mapped to a numeric score and combined with VADER through a gated fusion strategy to obtain a fused score and fused label. This hybrid design aims to preserve the speed and interpretability of lexicon-based scoring while adding structured, domain-specific evidence extraction for owner-oriented reporting.

Data mining is implemented as a per-review processing loop that combines a fast baseline sentiment method with structured evidence extraction. For each review, the pipeline first applies lexicon-based sentiment scoring (VADER) and then queries a local Ollama-compatible LLM to extract domain evidence and an overall sentiment label. The loop is executed inside a run-scoped folder (`RUN_DIR`), and all intermediate and final artifacts are written there to ensure traceability across executions.

5.5.1 VADER scoring

After preprocessing, the cleaned review text is passed to VADER to compute the *compound* sentiment score. The compound score is mapped into a bucket label (Positive/Neutral/Negative) using the thresholds defined in the algorithms chapter. Both the numeric score and the bucket label are stored in the processed dataset because they are used in multiple later stages: (i) aggregations for sentiment distribution, (ii) comparisons against LLM-based labels, and (iii) as one input for the fusion mechanism.

5.5.2 LLM extraction (strict JSON)

In parallel to VADER scoring, the pipeline queries a local LLM through an Ollama-compatible HTTP endpoint. The LLM is used to produce owner-oriented structured signals that are not available from VADER alone.

For each review, the extraction step returns domain-specific evidence in a consistent schema:

- **Kitchen:** short pros/cons evidence (or `n/a`),
- **Service:** short pros/cons evidence (or `n/a`),
- **Management:** short pros/cons evidence (or `n/a`),
- **Overall sentiment label:** Positive/Neutral/Negative (with confidence if available).

To ensure deterministic parsing, the prompts enforce strict JSON output. The pipeline parses the JSON response and writes the extracted fields into structured columns in the processed dataset. If strict parsing fails, a fallback parsing strategy is applied so that essential fields can still be recovered and the run can continue without losing the review-level output. This design choice is important for robustness because LLM responses can occasionally deviate from the expected format.

5.5.3 Fusion of VADER and LLM signals

To obtain a single decision-oriented sentiment result, the pipeline combines the VADER score with the LLM-derived sentiment signal into a fused score and fused label. The fusion is implemented as a gated weighting scheme: when VADER is close to neutral (low absolute magnitude), the pipeline assigns higher weight to the LLM; otherwise, the fusion slightly favors VADER. The fused score is written to the processed dataset together with the fused label used in reporting and visualization. This step reduces instability in borderline cases while preserving the interpretability of the baseline signal.

Listing 5.1: Simplified fused sentiment logic (from the implementation)

```
if abs(vader) < gate:
    w_v, w_l = 0.25, 0.75
else:
    w_v, w_l = 0.55, 0.45
fused_score = (w_v * vader) + (w_l * llm_score)
```

5.5.4 Resume-safe caching and run continuity

To support long runs and larger datasets, intermediate results are written periodically into `cache_reviews.csv` inside the run folder. The cache stores already processed review indices and their computed fields (VADER results, LLM extractions, and fused values). When the pipeline is restarted, it detects the cache and skips previously processed entries, continuing from the last completed index. This reduces re-processing time and makes the system robust against interruptions (e.g., system restarts or temporary LLM unavailability) while keeping outputs consistent across repeated executions.

5.5.5 Consolidation of outputs for downstream stages

After all reviews are processed, the pipeline consolidates the accumulated review-level results into the final processed dataset (e.g., `reviews_processed.csv`, and compatibility variants when enabled). These outputs are then consumed by subsequent stages of the product:

- **Owner summary generation:** evidence lines are selected and summarized into `owner_summary.json` and a readable text version,
- **Chart creation:** star distribution, sentiment distribution, and (if dates exist) trend charts are generated,
- **Email reporting:** helper scripts assemble the email body and attachments and send the weekly report via SMTP.

This ensures that the data mining loop not only produces sentiment labels, but also provides the structured evidence and artifacts required for an end-to-end owner-ready reporting workflow.

Part V

Development to Deployment

Chapter 6

Implementation Framework

6.1 Pipeline flow

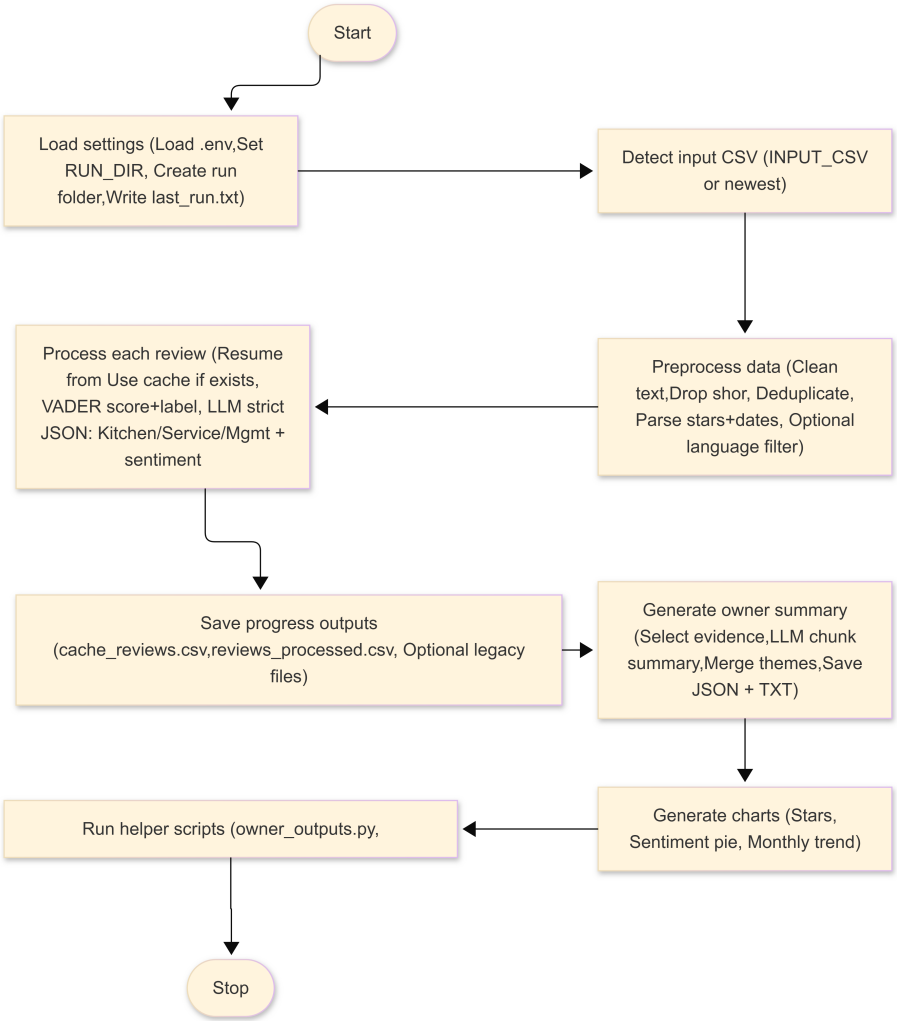

Figure 6.1: End-to-end workflow of the review analytics pipeline (created by authors).

Figure 6.1 summarizes the end-to-end workflow of the pipeline (one execution run). The workflow is designed as a repeatable run-based process: each execution creates (or reuses) a dedicated run folder so that all generated artifacts (processed data, summaries, charts, and email outputs) remain grouped and traceable. Input handling is configurable via the `.env` file and supports both fixed experiments (explicit `INPUT_CSV`) and practical usage (automatic selection of the newest available dataset).

After loading the dataset, the pipeline performs preprocessing steps such as text normalization, removal of very short entries, deduplication, and parsing of stars and dates. The main analysis then combines lexicon-based sentiment scoring (VADER) with LLM-based extraction to derive both quantitative sentiment signals and qualitative evidence quotes in the domains Kitchen, Service, and Management. To improve robustness and reproducibility, intermediate results are written incrementally to cache files so that interrupted runs can resume without reprocessing completed reviews. Finally, the pipeline generates owner-oriented outputs including structured summary files, visualizations, and an email-ready report that can be sent with attachments using the implemented email module.

6.2 Configuration

The system is designed to run end-to-end without modifying source code by externalizing all runtime settings into a `.env` file. This configuration layer defines the business context (e.g., `RESTAURANT_NAME`), input handling (either a fixed `INPUT_CSV` or automatic detection of the most recent suitable CSV in the project folder), and operational constraints such as `MAX_REVIEWS`, `MIN_TEXT_LEN`, and optional language filtering (`LANGUAGE`). The same file also contains the local LLM access parameters (`MODEL_NAME` and `LLM_URL`) to allow switching models or endpoints without code changes. For automated report delivery, SMTP credentials and recipients are configured in `.env` as well. This separation of configuration from implementation enables reproducible runs across machines, simplifies deployment, and supports quick adaptation to new restaurants or competitor datasets.

6.3 Per-run reproducibility

Each execution of the main pipeline is treated as an isolated run. A new run directory is created under `runs/YYYY-MM-DD_HHMMSS/`, and the path of the latest run is stored in `runs/last_run.txt`. This run-based design ensures that outputs from different executions are separated, comparable, and traceable. It also allows helper scripts to reliably locate the newest artifacts without manual path updates.

In addition to isolating outputs, the pipeline supports resume-safe operation through the cache mechanism. Intermediate results are written into `cache_reviews.csv` inside the current run folder. If a long run is interrupted (e.g., system restart or temporary LLM unavailability), the pipeline can continue from the cached progress and skip already processed reviews, which reduces runtime and prevents duplicate processing.

6.4 Pipeline Execution and Run Folder Management

Operation is organized around a per-run output directory concept to ensure traceability and to avoid mixing artifacts from different executions. When the main pipeline starts, it resolves a `RUN_DIR` as follows: if `RUN_DIR` is already provided as an environment variable, it is reused; otherwise, a new run folder is created under `runs/` using a timestamp-based name. The selected folder is written to `runs/last_run.txt` and exported back to the environment so that all helper scripts work on the same run context. This design guarantees that every run has its own audit trail containing the processed dataset, intermediate cache files, charts, summaries, and the final email body.

After establishing the run folder, the pipeline loads the input CSV, validates required columns, and performs preprocessing (cleaning, duplicate removal, date parsing, star normalization, and optional language filtering). During extraction, results are written repeatedly into a cache file inside the run directory to support recovery if execution is interrupted. Once processing completes, the final processed dataset and all reporting artifacts are produced and stored in the same run directory. The full run folder therefore represents a complete, reproducible snapshot of one execution.

6.5 From local development to repeatable execution

During development, the project is executed locally as a Python application with dependencies managed via `requirements.txt`. Configuration is externalized into `.env` so that environment-specific parameters (e.g., input file, language mode, model endpoint, SMTP settings) can be changed without code modifications. This separation of code and configuration supports a clean transition from development runs to operational runs.

Operational execution is performed by running the main script (pipeline) first and then running helper scripts that convert artifacts into owner-ready deliverables. Since all generated files are written into `RUN_DIR`, deployment does not require a database or external storage service. The run folder acts as a lightweight execution record and can be archived or shared as a complete result package.

6.6 Owner-facing artifacts

After the pipeline finishes, helper scripts convert the structured outputs into owner-facing deliverables. In particular, the JSON owner summary and the processed CSV are transformed into formats that are directly usable by non-technical stakeholders and business tools:

- a readable owner summary text (`owner_summary_readable.txt`) that highlights key themes and actions,
- a flattened summary CSV (`owner_summary_flat.csv`) suitable for BI tools or spreadsheet workflows,
- and an email body text (`weekly_owner_email.txt`) used for weekly reporting.

Along with these deliverables, the run includes chart images (stars distribution, sentiment distribution, monthly trend) and the full processed dataset (`reviews_processed.csv`), enabling owners to both read the summary and verify it against evidence.

6.7 Email delivery as deployment step

Email delivery is an integral part of the product workflow and supports regular, automated report distribution. The system resolves the latest successful run using `RUN_DIR` or `runs/last_run.txt` and relies on SMTP configuration provided via `.env`. At the end of a successful pipeline execution, a prepared email body (`weekly_owner_email.txt`) and all required reporting artifacts, including summaries, charts, and processed CSV files, are generated and stored within the run directory to ensure traceability. The `send_weekly_report.py` script then loads SMTP and recipient settings, validates and collects available attachments, and sends one email per configured recipient using the specified SMTP server. This design enables reliable, repeatable weekly reporting with minimal manual intervention while preserving a clear audit trail for each execution.

Part VI

Monitoring

Chapter 7

Monitoring

Monitoring in this project is designed to ensure operational visibility, traceability, and reproducibility across the entire analytics pipeline. Since the system integrates multiple stages (data loading, preprocessing, sentiment scoring, LLM-based extraction, aggregation, visualization, and email delivery), monitoring must support both technical debugging and business-level auditability. The implemented approach combines console-based observability with a per-run artifact trail, so that every execution can be inspected and reproduced without ambiguity.

7.1 Monitoring Goals

The monitoring concept follows four main goals. First, the system must remain traceable, meaning it must be possible to identify which input data and configuration produced a given output. Second, the system must be reproducible, such that repeated runs with the same inputs and configuration yield comparable outputs. Third, monitoring must provide strong debuggability, allowing developers to quickly isolate failures in preprocessing, model inference, or reporting. Finally, the monitoring approach must contribute to operational robustness, ensuring that temporary failures do not corrupt results or force a full re-run from scratch.

7.2 Runtime Logging and Console Observability

The pipeline provides step-by-step runtime observability through structured console output. At startup, the system prints an execution banner containing the restaurant name, the configured model name, and the resolved run folder path. Input resolution is also explicitly logged; the system prints which CSV file was selected, whether it was taken from `INPUT_CSV` or detected automatically. During preprocessing, the console output reports the review count after cleaning and filtering steps, which helps verify data quality and ensures that downstream analysis is performed on the intended subset of reviews.

For long-running executions, especially when LLM calls are made repeatedly, progress must be visible. The processing loop provides continuous progress reporting using a progress bar, enabling practical monitoring of runtime behavior and early detection of stalls. After every major stage, the system prints confirmation of saved outputs (processed CSV, owner summary, charts), which allows users to validate the workflow without manually checking intermediate directories.

7.3 Per-Run Output Folder as Audit Trail

A key monitoring feature is the per-run output folder design. Rather than writing outputs into a shared directory where artifacts can be overwritten or mixed, the pipeline creates a unique run directory for each execution. This folder typically follows the naming scheme `runs/YYYY-MM-DD_HHMMSS`, and it stores all artifacts generated during the run, including cached intermediate results, processed datasets, charts, owner summaries, and email drafts. In addition, the system writes the latest run folder path into `runs/last_run.txt`, allowing helper scripts to consistently locate the correct run outputs. This design acts as an audit trail by preserving the complete artifact set of each run and preventing accidental file duplication across different executions.

7.4 Checkpointing and Resume Support

A common failure mode in review analytics pipelines is the interruption of long runs, especially when hundreds of LLM calls are made. To prevent wasted runtime and repeated computation, the system implements checkpointing through a cache file stored inside the run folder. During processing, intermediate results are written into `cache_reviews.csv` at regular intervals. If a run is restarted, the pipeline checks whether the cache exists and resumes from the last processed review index, skipping already computed items. This monitoring-oriented design improves robustness and allows stable operation even in the presence of temporary crashes, user interruptions, or model service restarts.

7.5 Input Validation and Data Quality Monitoring

Monitoring also includes safeguards at the data level. Before analysis begins, the pipeline validates that review text content is available by checking the `review_text` column and attempting common fallback column names if needed. Ratings are normalized and clamped to a valid 1–5 scale, while timestamps are parsed into standardized date formats. Reviews below a minimum text length are discarded, and duplicates are removed to avoid biased statistics caused by repeated entries in exports. Optional language filtering further improves analytical consistency when the dataset contains multilingual reviews. These checks provide monitoring of input quality and reduce the risk of producing misleading outputs due to malformed or noisy data.

7.6 LLM Reliability Monitoring and JSON Safety

The LLM integration is the most sensitive part of the system, both in runtime stability and output correctness. For this reason, the pipeline includes explicit reliability monitoring for LLM calls. Each model request is executed with a timeout, preventing indefinite blocking. Failures trigger a retry mechanism with a controlled backoff, and warnings are printed to the console, making failures visible rather than silent. To enforce deterministic integration, the prompts require strict JSON outputs and define narrow extraction scopes (Kitchen, Service, Management, and an overall sentiment label). If the returned output is not valid JSON, the system attempts to recover JSON-like fragments using a parser fallback. This monitoring ensures that downstream aggregation operates on structured, validated data instead of unreliable free-form text.

7.7 Artifact Completeness and Output Verification

From a monitoring standpoint, a run is considered complete when the expected artifacts exist in the run directory. The system generates a processed dataset (CSV) containing per-review sentiment and extracted evidence fields, a structured owner summary (JSON) and a readable text version, and visualization charts for ratings distribution, sentiment distribution, and time trends. Because these artifacts form a consistent output contract, both developers and client users can validate run success by verifying the presence of the complete artifact set. This also supports operational debugging, as missing artifacts directly indicate which step failed or was skipped.

7.8 Email Delivery Monitoring

The weekly report delivery is treated as a monitored operational step rather than a side feature. Before sending emails, the script prints the SMTP configuration status and the list of recipients resolved from the environment file. It then lists the attachments found in the run directory, making it transparent what will be delivered to the owner. Emails are sent per recipient, and a success/failure status is printed for each recipient individually. In case of SMTP errors, exceptions are printed to the console and the final return code reflects failure, ensuring that delivery problems are visible and can be acted upon.

7.9 Recommended Monitoring Enhancements

While the current monitoring approach is suitable for a client-ready prototype, production deployments typically require structured observability beyond console output. A recommended enhancement is writing a structured log file (for example `run.log`) into each run folder, including timestamps and log levels. Another improvement is generating a `run_manifest.json` that records metadata such as model name, input file name, number of reviews processed, and runtime duration. Pre-flight health checks could also improve reliability by verifying that the LLM API is reachable and that SMTP authentication is valid before executing expensive processing steps. Finally, threshold-based warnings and alerting could be added to detect abnormal patterns such as unusually high negative sentiment, excessive review drops during filtering, or a large proportion of missing star ratings.

Part VII

Verification - Evaluation - Conclusion

Chapter 8

Evaluation / Verification

8.1 Evaluation Performed and Findings

The evaluation of this project focused on verifying that the generated outputs are (1) logically consistent, (2) traceable to the original reviews, and (3) useful for decision-making. The tests were executed directly on the produced pipeline outputs (processed CSV, evidence fields, and owner summary), making the verification reproducible for any new dataset.

8.1.1 Star rating vs. sentiment sanity check (Finding)

As a first verification, the star ratings (when available) were compared against both sentiment signals (VADER bucket and LLM label). In most cases, low-star reviews aligned with negative sentiment and high-star reviews aligned with positive sentiment, which indicates the pipeline behaves realistically. However, several mismatches were observed. These were not treated as pipeline errors, but as meaningful edge cases:

Some reviews contained mixed feedback (praise + complaints), resulting in neutral or mixed sentiment despite high/medium stars. A few reviews showed rating-text inconsistency (e.g., a higher star rating but strongly negative wording), demonstrating why text-based sentiment adds value beyond star averages.

8.1.2 Evidence grounding and traceability (Finding)

A second verification step validated whether the extracted evidence for *Kitchen, Service, Management* was grounded in the review text. A representative spot-check was performed across positive, neutral, and negative reviews. In the majority of checked cases, the evidence was either an exact phrase from the review or a faithful short excerpt, which supports trust in the reporting layer and owner summary.

A smaller number of cases required attention: Evidence occasionally became too generic (not clearly tied to a specific phrase). When a review mentioned multiple aspects in one sentence, evidence could be assigned to the wrong category (e.g., service complaint appearing under management).

Suggestion: Enforce stricter evidence rules (e.g., "return a short quote from the review text") and validate evidence spans by checking that the extracted phrase appears in the source text. Add a fallback that outputs "N/A" if the model cannot find direct evidence.

8.1.3 Agreement across sentiment signals (Finding)

The third verification tested internal agreement between VADER, LLM sentiment label, and the consolidated sentiment used in reporting. Overall agreement was observed frequently, which increases confidence in the final results. Disagreements appeared primarily in predictable scenarios:

Mixed-language reviews or non-English phrasing reduced VADER reliability. Reviews with sarcasm, short slang, or weak emotional words sometimes produced different labels across methods. Borderline reviews (balanced pros/cons) often split between neutral and negative depending on the model interpretation.

Suggestion: Log and monitor a "disagreement rate" over time. If disagreement rises, prioritize (a) language detection/translation, (b) prompt tightening, or (c) upgrading to a stronger model for uncertain cases.

8.1.4 Operational robustness checks (Finding)

In addition to content correctness, operational behavior was verified through repeated runs using caching/resume outputs. The pipeline produced stable files and consistent aggregation results when re-run on the same input, indicating reproducibility of the processing and reporting flow.

Suggestion: Add schema validation (e.g., JSON Schema) on every LLM response and introduce an automatic repair step for malformed outputs. This would reduce manual intervention and improve unattended runs.

8.2 Summary of Improvements for Stronger Verification

Based on the above findings, the most impactful next improvements are: Build a small labeled set (100–300 reviews) to compute quantitative accuracy and regression tests. Strengthen grounding: enforce quote-based evidence and validate that evidence exists in the review text.Improve language handling (language detector and optional translation).Optimize prompting and reduce cost/latency by calling the LLM only once per review (single structured extraction request instead of multiple calls).Monitor mismatch and disagreement rates as quality KPIs for production-style deployment.

8.3 Limitations

The evaluation must consider the limitations of each component of the pipeline. The VADER sentiment method is lexicon-based, which makes it fast and interpretable, but also sensitive to language and context. It may mis-handle sarcasm, irony, slang, or domain-specific language, and it performs best on English short texts. In multilingual datasets or German-heavy review collections, VADER results can become less reliable without translation or a language-specific sentiment approach.

LLM-based extraction improves semantic understanding and aspect separation, but it introduces its own uncertainties. Output quality depends on the selected local model, the prompt constraints, and the model's ability to follow strict JSON formatting. Even with strict prompting, occasional formatting errors or overly general responses can occur, and repeated calls across many reviews can amplify runtime and resource constraints.

Language filtering is implemented using heuristics rather than a dedicated language detection library. This is sufficient for medium and long reviews, but very short reviews such as "Gut" or "Nice" contain too little information to classify reliably. As a result, short reviews may be incorrectly filtered in or out, which can slightly affect sentiment distributions and summary coverage.

"'latex

Chapter 9

Future Work

The current version delivers a stable end-to-end pipeline, but several improvements would strengthen it further for production-like deployment. A practical next step is creating a small gold-standard dataset (e.g., 100–300 manually labeled reviews) for quantitative benchmarking and regression testing when prompts, thresholds, or models change.

Language handling can be improved by replacing heuristic checks with a dedicated language detector and optionally adding translation for mixed German/English datasets. This would make comparisons fairer across restaurants and platforms.

The LLM layer can be optimized in two directions: quality and efficiency. Using a stronger model (or an optional cloud LLM) may improve aspect extraction and sentiment consistency. In parallel, prompt optimization and schema tightening can reduce invalid outputs and improve evidence precision. A key engineering goal is to call the LLM only once per review by merging currently separated extraction steps into a single structured request, reducing latency and cost while simplifying the pipeline.

Finally, the reporting outputs can be extended from periodic emails to continuous monitoring by building a dashboard (e.g., Grafana) that tracks sentiment trends, aspect KPIs, and recurring themes over time, with optional alerts for sudden spikes in negative feedback. “‘

Chapter 10

Conclusion

The developed system demonstrates a practical local-first workflow for customer review analytics. It combines an interpretable baseline sentiment method (VADER) with controlled LLM-based evidence extraction and structured summarization. The result is a reproducible pipeline that converts unstructured review text into owner-ready outputs such as processed reports, visual trend charts, and an actionable executive summary grounded in review evidence. This approach supports both self-monitoring of a business and competitive benchmarking, and it can be generalized beyond restaurants to other product- and service-based domains where customer feedback plays a key role in continuous improvement.

Part VIII

Appendix

Appendix A

Configuration and environment file

A.1 Purpose of the .env file

The project is configured via a `.env` file placed in the project root. This keeps credentials (SMTP) and runtime parameters (LLM endpoint, model name, input CSV) out of the source code and supports reproducible execution across different machines.

A.2 Example .env template

Listing A.1 shows the provided template `.env.example` (included in the submission package).

Listing A.1: Example configuration file (`.env.example`)

```
# --- Email / SMTP ---
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=your.email@gmail.com
SMTP_PASS=YOUR_APP_PASSWORD
OWNER_EMAIL=owner@example.com
# or multiple recipients:
# OWNER_EMAILS=a@x.com,b@y.com

# --- LLM (Ollama / local) ---
LLM_URL=http://localhost:11434/api/generate
MODEL_NAME=phi3.5

# --- Restaurant / input ---
RESTAURANT_NAME=Pizza House
INPUT_CSV=sample_data/Burger King Data.csv
# Optional controls
# MAX_REVIEWS=50
# RUN_DIR=runs

# --- Optional language/filter settings ---
# LANGUAGE=auto # auto|de|en
# SUMMARY_LANGUAGE=auto # auto|de|en
# MIN_TEXT_LEN=10
```

```
25
26 # --- Compatibility ---
27 # WRITE_LEGACY_OUTPUTS=1 # also write older output filenames if required
```

A.3 Configuration notes

- **Email (SMTP)**: for Gmail, an "app password" is typically required (not the normal account password).
- **Recipients**: use `OWNER_EMAIL` for a single recipient or `OWNER_EMAILS` for a comma-separated list.
- **LLM runtime**: `LLM_URL` points to the local Ollama-compatible endpoint; `MODEL_NAME` selects the model.
- **Input selection**: set `INPUT_CSV` to pin a specific dataset; otherwise the newest CSV is selected automatically.

Appendix B

Output artifacts and file formats

B.1 Per-run folder structure

Each execution writes to a dedicated run folder: `runs/YYYY-MM-DD_HHMMSS/`. A pointer file `runs/last_run.txt` stores the latest run path for helper scripts.

B.2 Artifacts created in a run

Table B.1 summarizes the core artifacts written into the run folder.

Table B.1: Run artifacts and their purpose

File	Purpose
`cache_reviews.csv`	Resume file written during processing (checkpoint every 5 reviews).
`reviews_processed.csv`	Final per-review dataset with derived fields (VADER, LLM, fused).
`owner_summary.json`	Owner summary (structured JSON) created from evidence lines via
`owner_summary_readable.txt`	Human-readable representation of the JSON summary.
`owner_summary_flat.csv`	Flattened summary for spreadsheets/BI tools.
`chart_stars.png`	Star rating distribution chart.
`chart_sentiment_pie.png`	Sentiment distribution chart (fused labels if available).
`chart_trend.png`	Monthly sentiment trend (average score by month).
`weekly_owner_email.txt`	Email body produced by `owner_outputs.py`.

B.3 Processed CSV schema (high level)

The file `reviews_processed.csv` contains both the original review text and derived attributes used for analysis. Key columns include:

- identifiers: `idx`
- metadata: `date`, `stars`
- text: `review_text`
- VADER: `vader_score`, `sentiment_bucket`
- LLM sentiment: `llm_label`, `llm_confidence`, `llm_score`

- fusion: `fused_label`, `fused_score`, `fused_w_vader`, `fused_w_llm`
- evidence fields: `kitchen_pros/cons`, `service_pros/cons`, `mgmt_pros/cons`

B.4 Owner summary JSON format

The owner summary JSON follows the strict schema defined in the prompt (see Appendix C). Listing B.1 shows an *illustrative* example following the schema (values depend on the dataset).

Listing B.1: Illustrative owner summary JSON (schema example)

```
{
  "headline": "Overall feedback is positive with a few recurring issues to
     address.",
  "areas_of_excellence": [
    {
      "theme": "Fresh taste",
      "area": "Kitchen",
      "what_to_keep": "Customers repeatedly praise taste and freshness.",
      "how_to_market_it": "Highlight signature dishes and freshness in posts.",
      "evidence": ["\"Very tasty pizza\"", "\"Fresh ingredients\""]
    }
  ],
  "areas_of_improvement": [
    {
      "theme": "Waiting time",
      "area": "Service",
      "what_to_fix": "Some customers mention long waiting times during peak hours
         .",
      "suggested_action": "Add staffing or improve queue handling at peak times."
         ,
      "evidence": ["\"Waited 40 minutes\"", "\"Service was slow\""]
    }
  ],
  "quick_wins_next_7_days": [
    "Adjust staffing on peak days.",
    "Respond to recent negative reviews with concrete actions."
  ],
  "longer_term_30_days": [
    "Review workflow and training for peak-time processes."
  ],
  "suggested_kpis": [
    "Average star rating",
    "Fused sentiment score trend",
    "Count of waiting-time complaints"
  ]
}
```

B.5 Cache files and resume capability

The cache file `cache_reviews.csv` is a checkpoint written every 5 processed reviews. On restart, the pipeline loads the cache and skips already processed indices (`idx`), enabling:

- safe recovery from interruptions (power loss, network issues),
- incremental processing for large exports,
- reproducible comparison between different runs and models.

B.6 Big data handling considerations

The current implementation reads the input CSV into memory via `pandas.read_csv`, which is sufficient for typical restaurant exports. For very large datasets (hundreds of thousands to millions of reviews), the project already includes safeguards (resume cache and `MAX_REVIEWS`). Further scaling options (future work) include:

- streaming CSV processing via `chunksize` in `read_csv`,
- persisting intermediate results in a database (SQLite/PostgreSQL),
- batching LLM calls and applying backpressure based on runtime throughput,
- separating offline processing from report generation.

Appendix C

LLM prompts, model configuration, and prompt engineering

C.1 Why strict JSON prompts

The pipeline uses "strict JSON" prompts to make outputs machine-parsable and to reduce the risk of hallucinated content. Hard rules in the prompts include: *do not invent facts*, return only JSON, and use `n/a` when no evidence exists.

C.2 Prompts used in the implementation

Listing C.1 shows the exact prompt functions used for Kitchen/Service/Management extraction and sentiment labeling.

Listing C.1: Prompt functions used for evidence extraction and sentiment labeling (from the main pipeline)

```
def get_kitchen_prompt(text, stars):
    return f"""
Du bist ein Daten-Analyst. Extrahiere FAKTEN über das ESSEN.

Regeln:
1. Suche NUR nach: Geschmack, Temperatur, Frische, Portionen, Belag/Toppings,
    Zubereitung (zu trocken/zu dunkel/zu roh).
2. Ignoriere: Service, Preis, Wartezeit.
3. Wenn nichts erwähnt wird, antworte mit "n/a".
4. Kopiere kurze Zitate aus dem Text. Erfinde nichts.
5. Sterne sind nur Metadaten. Interpretiere sie NICHT. Nutze nur den Text.

Antworte als JSON:
{{
  "kitchen_pros": "Kurzes positives Zitat (oder 'n/a')",
  "kitchen_cons": "Kurzes negatives Zitat (oder 'n/a')"
}}

Sterne: {stars}
Text: "{text}"
""".strip()
```

```
21
22
23 def get_service_prompt(text, stars):
24     return f"""
25 Du bist ein Daten-Analyst. Extrahiere FAKTEN über SERVICE & LIEFERUNG.
26
27 Regeln:
28 1. Suche nach: Wartezeit, Fahrer, Freundlichkeit, Bestellgenauigkeit,
       Kommunikation, Sauberkeit.
29 2. Ignoriere: Geschmack des Essens.
30 3. Kopiere kurze Zitate. Erfinde nichts.
31 4. Wenn nichts erwähnt wird, antworte mit "n/a".
32 5. Sterne sind nur Metadaten. Interpretiere sie NICHT. Nutze nur den Text.
33
34 Antworte als JSON:
35 {{
36   "service_pros": "Kurzes positives Zitat (oder 'n/a')",
37   "service_cons": "Kurzes negatives Zitat (oder 'n/a')"
38 }}
39
40 Sterne: {stars}
41 Text: "{text}"
42 """.strip()
43
44
45 def get_mgmt_prompt(text, stars):
46     return f"""
47 Du bist ein Manager. Extrahiere FAKTEN über MANAGEMENT & PREIS.
48
49 Regeln:
50 1. Suche nach: Preis/Leistung, Ambiente, Sauberkeit/Organisation,
       Konkurrenzvergleich (z.B. McDonald's), Verbesserungsvorschläge.
51 2. Ignoriere Essen/Service (außer wenn es strategische Hinweise enthält, z.B. "
       zu teuer", "schlechte Organisation").
52 3. Kopiere kurze Zitate. Erfinde nichts.
53 4. Wenn nichts erwähnt wird, antworte mit "n/a".
54 5. Sterne sind nur Metadaten. Interpretiere sie NICHT. Nutze nur den Text.
55
56 Antworte als JSON:
57 {{
58   "mgmt_pros": "Kurzes positives Zitat (oder 'n/a')",
59   "mgmt_cons": "Kurzes negatives Zitat (oder 'n/a')"
60 }}
61
62 Sterne: {stars}
63 Text: "{text}"
64 """.strip()
65
66
67 def get_sentiment_prompt(text, stars):
68     return f"""
```

```
69 Classify the overall sentiment based ONLY on the review text (ignore star rating
       ).
70 Return STRICT JSON with exactly these keys:
71 {{
72   "label": "positive" | "neutral" | "negative",
73   "confidence": 0.0-1.0
74 }}
75
76 Stars (context only, do NOT use for the decision): {stars}
77 Review: "{text}"
78 """.strip()
79
80
81 # ----------------------------
82 # OWNER SUMMARY HELPERS
83 # ----------------------------
84 def chunk_list(items, chunk_size=10):
85     for i in range(0, len(items), chunk_size):
86         yield items[i:i + chunk_size]
87
88
89 def build_evidence_lines(final_df: pd.DataFrame) -> list:
90     lines = []
91     for _, r in final_df.iterrows():
92         text = str(r.get("review_text", "")).strip()
93         stars = r.get("stars", "")
94         vader = r.get("vader_score", 0.0)
95         bucket = r.get("sentiment_bucket", "")
96
97         if len(text) > 340:
98             text = text[:340] + "..."
99
100        kp = str(r.get("kitchen_pros", "")).strip()
101        kc = str(r.get("kitchen_cons", "")).strip()
102        sp = str(r.get("service_pros", "")).strip()
103        sc = str(r.get("service_cons", "")).strip()
104        mp = str(r.get("mgmt_pros", "")).strip()
105        mc = str(r.get("mgmt_cons", "")).strip()
106
107        line = f'- Stars:{stars} | VADER:{float(vader):.2f} | {bucket} | Review
               :"{text}"'
108        extras = []
109        if kp:
110            extras.append(f"Kitchen+:{kp}")
111        if kc:
112            extras.append(f"Kitchen-:{kc}")
113        if sp:
114            extras.append(f"Service+:{sp}")
115        if sc:
116            extras.append(f"Service-:{sc}")
117        if mp:
```

```
            extras.append(f"Mgmt+:{mp}")
        if mc:
            extras.append(f"Mgmt-:{mc}")

        if extras:
            line += " | " + " | ".join(extras)

        lines.append(line)
    return lines


def chunk_summary_prompt(evidence_lines: list) -> str:
    joined = "\n".join(evidence_lines)
```

C.3 Owner summary prompt and schema

Listing C.2 shows the schema for chunk summaries and the final owner summary.

Listing C.2: Owner summary prompts and schema (from the main pipeline)

```
    return f"""
Du bist ein Restaurant-Berater. Du analysierst Kundenfeedback und extrahierst
    NUR belegbare Punkte aus den Evidence-Lines.

WICHTIG (harte Regeln):
- Erfinde nichts. Nutze nur die Evidence-Lines.
- Verwende kurze wörtliche Belege direkt aus den Lines (Evidence-Zitate).
- Sterne sind nur Metadaten. Interpretiere sie NICHT.
- Gib NUR gültiges JSON zurück (kein Markdown, keine Kommentare).
- Wenn du keinen Punkt belegen kannst: gib leere Listen zurück.
- Jeder Improvement-Punkt MUSS "what_to_fix" UND "suggested_action" haben.
- Jeder Excellence-Punkt MUSS "what_to_keep" haben.

Ausgabeformat (strikt):
{{
  "excellence": [
    {{
      "theme": "kurzer Titel",
      "area": "Kitchen|Service|Management",
      "what_to_keep": "1 Satz",
      "how_to_market_it": "optional",
      "evidence": ["Zitat 1", "Zitat 2"]
    }}
  ],
  "improvements": [
    {{
      "theme": "kurzer Titel",
      "area": "Kitchen|Service|Management",
      "what_to_fix": "1 Satz",
      "suggested_action": "1 Satz",
      "evidence": ["Zitat 1", "Zitat 2"]
```

```
    }}
  ]
}}

EVIDENCE-LINES:
{joined}
""".strip()


def final_owner_summary_prompt(merged: dict) -> str:
    blob = json.dumps(merged, ensure_ascii=False)
    return f"""
Du bist ein Restaurant-Owner-Advisor. Erstelle eine klare, gut strukturierte Zusammenfassung für den Restaurantinhaber.

Regeln:
- Nutze nur die Themen + Belege aus dem JSON.
- Keine neuen Fakten erfinden.
- Priorisiere die wichtigsten 3-5 Punkte je Kategorie.
- Gib konkrete Handlungsempfehlungen (Quick Wins vs. langfristig).
- Output strikt als JSON.

Format:
{{
  "headline": "...",
  "areas_of_excellence": [
    {{"theme":"...", "area":"...", "what_to_keep":"...", "how_to_market_it":"...", "evidence":["..."]}}
  ],
  "areas_of_improvement": [
    {{"theme":"...", "area":"...", "what_to_fix":"...", "suggested_action":"...", "evidence":["..."]}}
  ],
  "quick_wins_next_7_days": ["...","..."],
  "longer_term_30_days": ["...","..."],
  "suggested_kpis": ["...","..."]
}}

INPUT THEMES JSON:
{blob}
""".strip()


def merge_chunk_outputs(chunk_outputs: list) -> dict:
    merged = {"excellence": [], "improvements": []}
    seen = set()

    def norm(x: str) -> str:
        return re.sub(r"\s+", " ", str(x).strip().lower())

    def add_items(key: str, items):
```

```
79          if not isinstance(items, list):
80              return
81          for it in items:
```

C.4 Model selection and evolution

The implementation is model-agnostic: the model name is configured via `MODEL_NAME` in `.env`. In the submitted configuration template, `phi3.5` is used as a practical default for local execution. When documenting experiments, it is recommended to report:

- which models were tested locally (name and size),
- observed quality of strict JSON compliance,
- runtime and stability (timeouts / retries),
- and qualitative differences in extracted evidence.

Appendix D

Email reporting and attachments

D.1 Email generation vs. sending

Email reporting has two parts:

1. **Generation**: `owner_outputs.py` builds an owner-facing email text file (`weekly_owner_email.txt`) based on `owner_summary.json` and the processed dataset.
2. **Sending**: `send_weekly_report.py` reads the email text file and sends it via SMTP, attaching charts and summary files found in the run folder.

D.2 Deduplicated attachments

The sending script searches for a fixed set of expected artifact names (charts, summaries, processed CSV, cache) and attaches only those files that exist in the current run folder. This avoids duplicated email attachments when legacy filenames are also produced.

Appendix E

Illustrative output examples

The following figures show *possible* outputs produced by the pipeline. They are illustrative examples (created by the authors) to demonstrate how the charts and email appear; actual results depend on the input dataset.

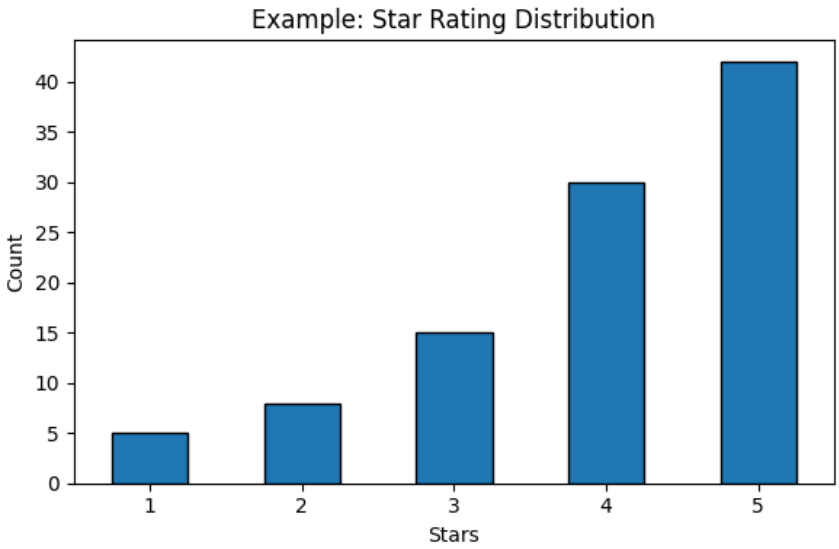

Figure E.1: Example output: star rating distribution chart (`chart_stars.png`).

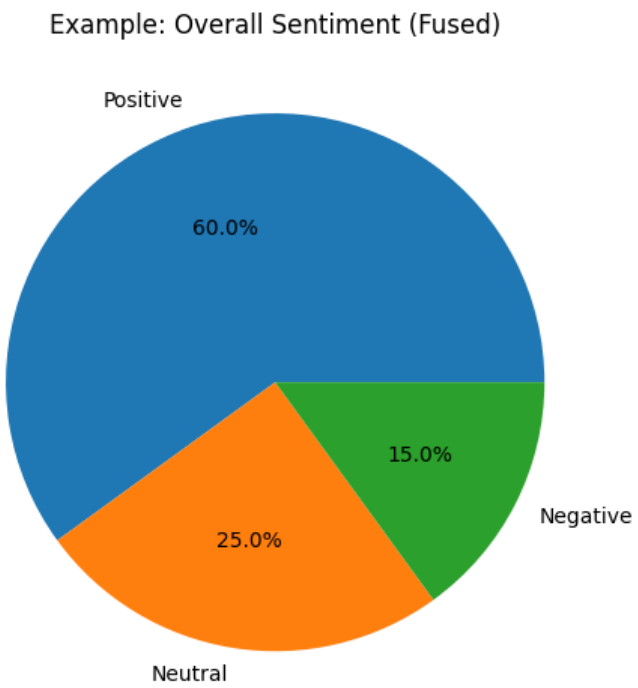

Figure E.2: Example output: sentiment pie chart (`chart_sentiment_pie.png`).

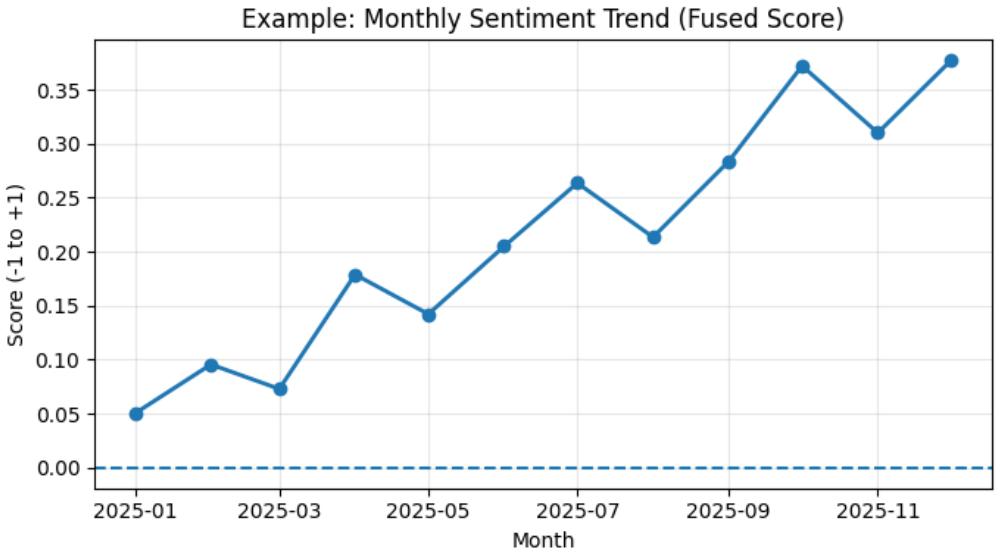

Figure E.3: Example output: monthly sentiment trend (`chart_trend.png`).

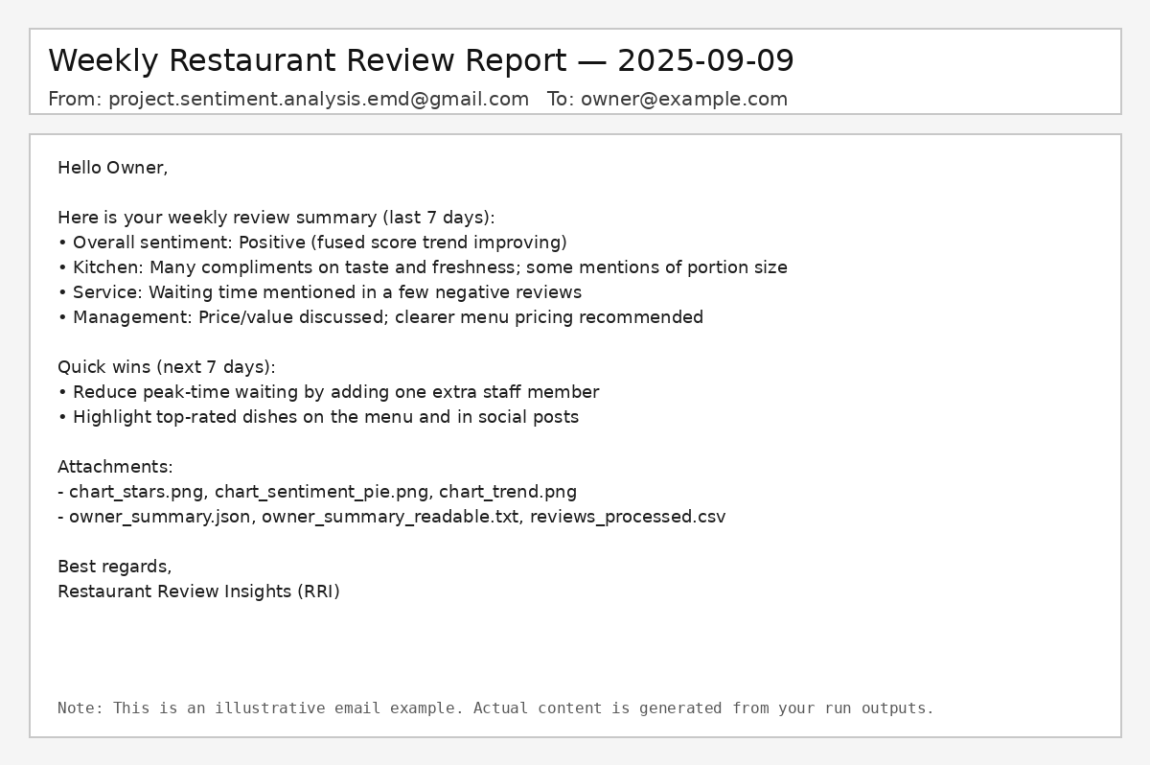

Figure E.4: Example output: weekly owner email (generated body + attachments).

Appendix F

Source code appendix (full listings with explanations)

This appendix includes the complete source code used for the product, together with short explanations per file (similar to the "Package Example" style in the reference report template).

F.1 Main pipeline: `Project_Sentiment_Analysis_22.12.1.py`

F.1.1 Description

The main script performs the end-to-end processing:

- resolves the run folder and selects the input CSV,
- preprocesses the data (cleaning, filtering, deduplication, parsing stars/dates),
- iterates reviews and extracts evidence with strict JSON prompts (Kitchen/Service/-Management) and sentiment labels,
- computes VADER and fused sentiment, writing progress to `cache_reviews.csv`,
- writes processed outputs, builds owner summary via chunking evidence lines,
- generates charts and prints a terminal summary.

F.1.2 Code

Listing F.1: Full main pipeline script

```
import os
import re
import time
import json
import requests
import pandas as pd
import nltk
import matplotlib.pyplot as plt
from tqdm import tqdm
from pathlib import Path
```

```
from nltk.corpus import stopwords
from nltk.sentiment import SentimentIntensityAnalyzer
from collections import Counter
from datetime import datetime


# ----------------------------
# PATHS (robust)
# ----------------------------
HERE = Path(__file__).resolve().parent

# ----------------------------
# RUN OUTPUT FOLDER (one folder per run)
# ----------------------------
RUNS_ROOT = HERE / "runs"

def get_run_dir() -> Path:
    """Create/resolve a per-run output folder and expose it via env vars.
    - If RUN_DIR is already set, reuse it.
    - Otherwise create runs/YYYY-MM-DD_HHMMSS and write runs/last_run.txt.
    """
    env_dir = os.getenv("RUN_DIR", "").strip()
    if env_dir:
        d = Path(env_dir)
        d.mkdir(parents=True, exist_ok=True)
        return d

    RUNS_ROOT.mkdir(parents=True, exist_ok=True)
    run_id = datetime.now().strftime("%Y-%m-%d_%H%M%S")
    d = RUNS_ROOT / run_id
    d.mkdir(parents=True, exist_ok=True)

    (RUNS_ROOT / "last_run.txt").write_text(str(d), encoding="utf-8")

    os.environ["RUN_DIR"] = str(d)
    os.environ["RUN_STARTED_AT"] = datetime.now().strftime("%Y-%m-%d %H:%M")
    return d


# ----------------------------
# ENV LOADER
# ----------------------------
def load_dotenv(path: Path, overwrite: bool = False):
    """Minimal .env loader (no extra packages). Loads key=value pairs."""
    if not path.exists():
        print(f"[ENV] .env not found at: {path}")
        return
    for line in path.read_text(encoding="utf-8").splitlines():
        line = line.strip()
        if not line or line.startswith("#") or "=" not in line:
            continue
```

```
        k, v = line.split("=", 1)
        k = k.strip()
        v = v.strip().strip('"').strip("'")
        if not k:
            continue
        if overwrite or (k not in os.environ):
            os.environ[k] = v


# Load env vars early so subprocess inherits them
load_dotenv(HERE / ".env", overwrite=False)

# ----------------------------
# SETTINGS (from env where possible)
# ----------------------------
RESTAURANT_NAME = os.getenv("RESTAURANT_NAME", "").strip() or "Restaurant"
MODEL_NAME = os.getenv("MODEL_NAME", "").strip() or "phi3.5"
LLM_URL = os.getenv("LLM_URL", "http://localhost:11434/api/generate").strip()

LANGUAGE = os.getenv("LANGUAGE", "auto").strip().lower() # auto|de|en
SUMMARY_LANGUAGE = os.getenv("SUMMARY_LANGUAGE", "auto").strip().lower()
MIN_TEXT_LEN = int(os.getenv("MIN_TEXT_LEN", "10").strip() or "10")

MAX_REVIEWS_ENV = os.getenv("MAX_REVIEWS", "").strip()
MAX_REVIEWS = int(MAX_REVIEWS_ENV) if MAX_REVIEWS_ENV.isdigit() else None

# Ensure helper scripts inherit these too
os.environ["MODEL_NAME"] = MODEL_NAME
os.environ["RESTAURANT_NAME"] = RESTAURANT_NAME
os.environ["LLM_URL"] = LLM_URL


# ----------------------------
# OUTPUT FILENAMES (generic + backward compatible)
# ----------------------------
CACHE_FILE_GENERIC = "cache_reviews.csv"
PROCESSED_FILE_GENERIC = "reviews_processed.csv"
OWNER_JSON_GENERIC = "owner_summary.json"
OWNER_TXT_GENERIC = "owner_summary_readable.txt"
OWNER_FLAT_GENERIC = "owner_summary_flat.csv"
EMAIL_TXT_GENERIC = "weekly_owner_email.txt"

# backward-compat names (also written)
CACHE_FILE_BK = "bk_cache_final.csv"
PROCESSED_FILE_BK = "bk_final_report.csv"
OWNER_JSON_BK = "bk_owner_summary.json"
OWNER_TXT_BK = "bk_owner_summary.txt"

CHART_STARS = "chart_stars.png"
CHART_SENTIMENT = "chart_sentiment_pie.png"
CHART_TREND = "chart_trend.png"
```

```
113
114 # backward-compat chart names
115 CHART1_BK = "chart_1_stars.png"
116 CHART2_BK = "chart_2_sentiment_pie.png"
117 CHART3_BK = "chart_3_trend.png"
118
119 # If you still need the old bk_* outputs, set WRITE_LEGACY_OUTPUTS=1 in .env
120 WRITE_LEGACY_OUTPUTS = os.getenv("WRITE_LEGACY_OUTPUTS", "0").strip().lower() in
      ("1", "true", "yes", "y")
121
122
123 # ----------------------------
124 # NLTK SAFE SETUP
125 # ----------------------------
126 def ensure_nltk_resources():
127     try:
128         _ = stopwords.words("german")
129     except LookupError:
130         nltk.download("stopwords", quiet=True)
131
132     try:
133         _ = SentimentIntensityAnalyzer()
134     except LookupError:
135         nltk.download("vader_lexicon", quiet=True)
136
137
138 ensure_nltk_resources()
139 sia = SentimentIntensityAnalyzer()
140 GERMAN_STOPS = set(stopwords.words("german"))
141 ENGLISH_STOPS = set(stopwords.words("english"))
142
143
144 # ----------------------------
145 # INPUT CSV AUTO-DETECT
146 # ----------------------------
147 def detect_input_csv() -> Path:
148     """Choose input CSV:
149     1) INPUT_CSV from env (absolute or relative to script folder)
150     2) else: choose the most recently modified CSV in script folder (excluding
          runs/ and obvious outputs)
151     """
152     env_csv = os.getenv("INPUT_CSV", "").strip()
153     if env_csv:
154         p = Path(env_csv)
155         if not p.is_absolute():
156             p = HERE / p
157         return p
158
159     candidates = []
160     for p in HERE.glob("*.csv"):
161         name = p.name.lower()
```

```
162         if name.startswith("cache_"):
163             continue
164         if "processed" in name or "owner_summary" in name or "weekly_owner_email"
                in name:
165             continue
166         if name.startswith("bk_"):
167             continue
168         candidates.append(p)
169 
170     # If nothing found, fallback to common legacy file if present
171     legacy = HERE / "Burger King Data.csv"
172     if legacy.exists():
173         return legacy
174 
175     if not candidates:
176         return legacy # will error later with a clear message
177 
178     # Most recently modified
179     candidates.sort(key=lambda x: x.stat().st_mtime, reverse=True)
180     return candidates[0]
181 
182 
183 # ----------------------------
184 # UTILS
185 # ----------------------------
186 def get_vader_score(text: str) -> float:
187     return sia.polarity_scores(str(text))["compound"]
188 
189 
190 def vader_bucket(score: float) -> str:
191     if score >= 0.05:
192         return "Positive"
193     if score <= -0.05:
194         return "Negative"
195     return "Neutral"
196 
197 
198 def is_german_review(text: str) -> bool:
199     """Heuristic German detection for filtering."""
200     t = str(text).lower()
201     if len(t) < 5:
202         return False
203     if any(ch in t for ch in "äöüß"):
204         return True
205     words = set(t.split())
206     return len(words.intersection(GERMAN_STOPS)) >= len(words.intersection(
            ENGLISH_STOPS))
207 
208 
209 def is_english_review(text: str) -> bool:
210     t = str(text).lower()
```

```
    words = set(t.split())
    # simple heuristic: more English stopwords than German
    return len(words.intersection(ENGLISH_STOPS)) > len(words.intersection(
        GERMAN_STOPS))


def parse_date(date_str):
    s = str(date_str).strip()
    if not s or s.lower() == "nan":
        return pd.NaT
    dt = pd.to_datetime(s, dayfirst=False, errors="coerce")
    if pd.notna(dt):
        return dt
    return pd.to_datetime(s, dayfirst=True, errors="coerce")


def clean_review_text(s: str) -> str:
    s = "" if s is None else str(s)
    s = s.strip()
    s = re.sub(r"\s+", " ", s)
    return s


def clamp_star(x):
    """Return int 1..5 or NaN."""
    try:
        v = float(str(x).replace("*", "").strip())
        if v != v:
            return pd.NA
        v = int(round(v))
        if v < 1 or v > 5:
            return pd.NA
        return v
    except Exception:
        return pd.NA


def call_llm(prompt, timeout=240, retries=3):
    """Ollama /api/generate compatible call. Change MODEL_NAME only in env."""
    payload = {
        "model": MODEL_NAME,
        "prompt": prompt,
        "stream": False,
        "temperature": 0.0,
        "format": "json",
    }
    last_err = None
    for attempt in range(1, retries + 1):
        try:
            r = requests.post(LLM_URL, json=payload, timeout=timeout)
            if r.status_code == 200:
```

```
                return r.json().get("response", "")
            last_err = f"HTTP {r.status_code}: {r.text[:200]}"
        except Exception as e:
            last_err = str(e)

        print(f"LLM Warning (attempt {attempt}/{retries}): {last_err}")
        time.sleep(2 * attempt)

    return "{}"


def parse_json_response(text):
    try:
        return json.loads(text)
    except Exception:
        match = re.search(r"(\{.*\})", str(text), re.DOTALL)
        if match:
            clean = match.group(1).replace("'", '"')
            try:
                return json.loads(clean)
            except Exception:
                pass
    return {}


def normalize_llm_label(label: str) -> str:
    """Normalize various label spellings/languages to positive/neutral/negative.
        """
    s = (label or "").strip().lower()
    # common German outputs
    if s.startswith("pos") or "positiv" in s:
        return "positive"
    if s.startswith("neg") or "negativ" in s:
        return "negative"
    if s.startswith("neu") or "neutral" in s:
        return "neutral"
    return "neutral"


def llm_label_to_score(label: str, conf: float) -> float:
    """Map LLM label+confidence to a score in [-1, +1]."""
    lab = normalize_llm_label(label)
    try:
        c = float(conf)
    except Exception:
        c = 0.6
    c = max(0.0, min(1.0, c))
    if lab == "positive":
        return +c
    if lab == "negative":
        return -c
```

```
    return 0.0


def fuse_sentiment(vader_compound: float, llm_label: str, llm_conf: float,
                   gate: float = 0.25,
                   pos_th: float = 0.10,
                   neg_th: float = -0.10):
    """
    Gated fusion of VADER (compound) and LLM (label+confidence).
    Returns: fused_label, fused_score, w_vader, w_llm
    """
    try:
        v = float(vader_compound)
    except Exception:
        v = 0.0

    l = llm_label_to_score(llm_label, llm_conf)

    # If VADER is unsure (close to 0), trust LLM more.
    if abs(v) < gate:
        w_v, w_l = 0.25, 0.75
    else:
        w_v, w_l = 0.55, 0.45

    fused_score = (w_v * v) + (w_l * l)

    if fused_score >= pos_th:
        fused_label = "Positive"
    elif fused_score <= neg_th:
        fused_label = "Negative"
    else:
        fused_label = "Neutral"

    return fused_label, fused_score, w_v, w_l


# ----------------------------
# STRICT PROMPTS
# ----------------------------
def get_kitchen_prompt(text, stars):
    return f"""
Du bist ein Daten-Analyst. Extrahiere FAKTEN über das ESSEN.

Regeln:
1. Suche NUR nach: Geschmack, Temperatur, Frische, Portionen, Belag/Toppings,
    Zubereitung (zu trocken/zu dunkel/zu roh).
2. Ignoriere: Service, Preis, Wartezeit.
3. Wenn nichts erwähnt wird, antworte mit "n/a".
4. Kopiere kurze Zitate aus dem Text. Erfinde nichts.
5. Sterne sind nur Metadaten. Interpretiere sie NICHT. Nutze nur den Text.

```

```
Antworte als JSON:
{{
  "kitchen_pros": "Kurzes positives Zitat (oder 'n/a')",
  "kitchen_cons": "Kurzes negatives Zitat (oder 'n/a')"
}}

Sterne: {stars}
Text: "{text}"
""".strip()


def get_service_prompt(text, stars):
    return f"""
Du bist ein Daten-Analyst. Extrahiere FAKTEN über SERVICE & LIEFERUNG.

Regeln:
1. Suche nach: Wartezeit, Fahrer, Freundlichkeit, Bestellgenauigkeit,
    Kommunikation, Sauberkeit.
2. Ignoriere: Geschmack des Essens.
3. Kopiere kurze Zitate. Erfinde nichts.
4. Wenn nichts erwähnt wird, antworte mit "n/a".
5. Sterne sind nur Metadaten. Interpretiere sie NICHT. Nutze nur den Text.

Antworte als JSON:
{{
  "service_pros": "Kurzes positives Zitat (oder 'n/a')",
  "service_cons": "Kurzes negatives Zitat (oder 'n/a')"
}}

Sterne: {stars}
Text: "{text}"
""".strip()


def get_mgmt_prompt(text, stars):
    return f"""
Du bist ein Manager. Extrahiere FAKTEN über MANAGEMENT & PREIS.

Regeln:
1. Suche nach: Preis/Leistung, Ambiente, Sauberkeit/Organisation,
    Konkurrenzvergleich (z.B. McDonald's), Verbesserungsvorschläge.
2. Ignoriere Essen/Service (außer wenn es strategische Hinweise enthält, z.B. "
    zu teuer", "schlechte Organisation").
3. Kopiere kurze Zitate. Erfinde nichts.
4. Wenn nichts erwähnt wird, antworte mit "n/a".
5. Sterne sind nur Metadaten. Interpretiere sie NICHT. Nutze nur den Text.

Antworte als JSON:
{{
  "mgmt_pros": "Kurzes positives Zitat (oder 'n/a')",
  "mgmt_cons": "Kurzes negatives Zitat (oder 'n/a')"
```

```
}}

Sterne: {stars}
Text: "{text}"
""".strip()


def get_sentiment_prompt(text, stars):
    return f"""
Classify the overall sentiment based ONLY on the review text (ignore star rating).
Return STRICT JSON with exactly these keys:
{{
  "label": "positive" | "neutral" | "negative",
  "confidence": 0.0-1.0
}}

Stars (context only, do NOT use for the decision): {stars}
Review: "{text}"
""".strip()


# ----------------------------
# OWNER SUMMARY HELPERS
# ----------------------------
def chunk_list(items, chunk_size=10):
    for i in range(0, len(items), chunk_size):
        yield items[i:i + chunk_size]


def build_evidence_lines(final_df: pd.DataFrame) -> list:
    lines = []
    for _, r in final_df.iterrows():
        text = str(r.get("review_text", "")).strip()
        stars = r.get("stars", "")
        vader = r.get("vader_score", 0.0)
        bucket = r.get("sentiment_bucket", "")

        if len(text) > 340:
            text = text[:340] + "..."

        kp = str(r.get("kitchen_pros", "")).strip()
        kc = str(r.get("kitchen_cons", "")).strip()
        sp = str(r.get("service_pros", "")).strip()
        sc = str(r.get("service_cons", "")).strip()
        mp = str(r.get("mgmt_pros", "")).strip()
        mc = str(r.get("mgmt_cons", "")).strip()

        line = f'- Stars:{stars} | VADER:{float(vader):.2f} | {bucket} | Review:"{text}"'
        extras = []
```

```
        if kp:
            extras.append(f"Kitchen+:{kp}")
        if kc:
            extras.append(f"Kitchen-:{kc}")
        if sp:
            extras.append(f"Service+:{sp}")
        if sc:
            extras.append(f"Service-:{sc}")
        if mp:
            extras.append(f"Mgmt+:{mp}")
        if mc:
            extras.append(f"Mgmt-:{mc}")

        if extras:
            line += " | " + " | ".join(extras)

        lines.append(line)
    return lines


def chunk_summary_prompt(evidence_lines: list) -> str:
    joined = "\n".join(evidence_lines)
    return f"""
Du bist ein Restaurant-Berater. Du analysierst Kundenfeedback und extrahierst
    NUR belegbare Punkte aus den Evidence-Lines.

WICHTIG (harte Regeln):
- Erfinde nichts. Nutze nur die Evidence-Lines.
- Verwende kurze wörtliche Belege direkt aus den Lines (Evidence-Zitate).
- Sterne sind nur Metadaten. Interpretiere sie NICHT.
- Gib NUR gültiges JSON zurück (kein Markdown, keine Kommentare).
- Wenn du keinen Punkt belegen kannst: gib leere Listen zurück.
- Jeder Improvement-Punkt MUSS "what_to_fix" UND "suggested_action" haben.
- Jeder Excellence-Punkt MUSS "what_to_keep" haben.

Ausgabeformat (strikt):
{{
  "excellence": [
    {{
      "theme": "kurzer Titel",
      "area": "Kitchen|Service|Management",
      "what_to_keep": "1 Satz",
      "how_to_market_it": "optional",
      "evidence": ["Zitat 1", "Zitat 2"]
    }}
  ],
  "improvements": [
    {{
      "theme": "kurzer Titel",
      "area": "Kitchen|Service|Management",
      "what_to_fix": "1 Satz",
```

```
      "suggested_action": "1 Satz",
      "evidence": ["Zitat 1", "Zitat 2"]
    }}
  ]
}}

EVIDENCE-LINES:
{joined}
""".strip()


def final_owner_summary_prompt(merged: dict) -> str:
    blob = json.dumps(merged, ensure_ascii=False)
    return f"""
Du bist ein Restaurant-Owner-Advisor. Erstelle eine klare, gut strukturierte Zusammenfassung für den Restaurantinhaber.

Regeln:
- Nutze nur die Themen + Belege aus dem JSON.
- Keine neuen Fakten erfinden.
- Priorisiere die wichtigsten 3-5 Punkte je Kategorie.
- Gib konkrete Handlungsempfehlungen (Quick Wins vs. langfristig).
- Output strikt als JSON.

Format:
{{
  "headline": "...",
  "areas_of_excellence": [
    {{"theme":"...", "area":"...", "what_to_keep":"...", "how_to_market_it":"...", "evidence":["..."]}}
  ],
  "areas_of_improvement": [
    {{"theme":"...", "area":"...", "what_to_fix":"...", "suggested_action":"...", "evidence":["..."]}}
  ],
  "quick_wins_next_7_days": ["...","..."],
  "longer_term_30_days": ["...","..."],
  "suggested_kpis": ["...","..."]
}}

INPUT THEMES JSON:
{blob}
""".strip()


def merge_chunk_outputs(chunk_outputs: list) -> dict:
    merged = {"excellence": [], "improvements": []}
    seen = set()

    def norm(x: str) -> str:
        return re.sub(r"\s+", " ", str(x).strip().lower())
```

```
556
557     def add_items(key: str, items):
558         if not isinstance(items, list):
559             return
560         for it in items:
561             if not isinstance(it, dict):
562                 continue
563             theme = str(it.get("theme", "")).strip()
564             area = str(it.get("area", "")).strip()
565             if not theme or not area:
566                 continue
567             sig = (norm(theme), norm(area))
568             if sig in seen:
569                 continue
570             seen.add(sig)
571             merged[key].append(it)
572
573     for out in chunk_outputs:
574         if not isinstance(out, dict):
575             continue
576         add_items("excellence", out.get("excellence", []))
577         add_items("improvements", out.get("improvements", []))
578
579     return merged
580
581
582 # ----------------------------
583 # MAIN
584 # ----------------------------
585 def main():
586     print("=" * 60)
587     print(f"RESTAURANT INSIGHTS PIPELINE -{RESTAURANT_NAME}")
588     print(f"Model: {MODEL_NAME}")
589     print("=" * 60)
590
591     os.chdir(HERE)
592     run_dir = get_run_dir()
593     print(f"[RUN FOLDER] {run_dir}")
594
595     # 1) LOAD DATA
596     input_csv = detect_input_csv()
597     if not input_csv.exists():
598         print(f"ERROR: Input CSV not found: {input_csv}")
599         print("Tip: set INPUT_CSV in .env or place a CSV in the script folder.")
600         return
601
602     print(f"[INPUT] Using CSV: {input_csv.name}")
603     df = pd.read_csv(input_csv)
604
605     # 2) PRE-PROCESS (generic + safe)
606     print("[1] Processing & Filtering...")
```

```
607
608     # Find review_text column
609     if "review_text" not in df.columns:
610         # try common alternatives
611         for alt in ["text", "review", "content", "comment", "reviews"]:
612             if alt in df.columns:
613                 df = df.rename(columns={alt: "review_text"})
614                 break
615
616     if "review_text" not in df.columns:
617         print("Error: 'review_text' column missing.")
618         print("Columns found:", list(df.columns))
619         return
620
621     # stars column discovery
622     star_col = None
623     for c in ["review_rating", "stars", "rating", "score"]:
624         if c in df.columns:
625             star_col = c
626             break
627
628     # date column discovery
629     date_col = None
630     for c in ["review_datetime_utc", "date", "timestamp", "created_at", "time"]:
631         if c in df.columns:
632             date_col = c
633             break
634
635     df["review_text"] = df["review_text"].astype(str).map(clean_review_text)
636     df = df[df["review_text"].str.lower() != "nan"].copy()
637     df = df[df["review_text"].str.len() >= MIN_TEXT_LEN].copy()
638
639     # remove duplicates (common in exports)
640     df = df.drop_duplicates(subset=["review_text"]).copy()
641
642     if date_col:
643         df["parsed_date"] = df[date_col].apply(parse_date)
644     else:
645         df["parsed_date"] = pd.NaT
646
647     if star_col:
648         df["stars"] = df[star_col].apply(clamp_star)
649     else:
650         df["stars"] = pd.NA
651
652     # language filtering
653     if LANGUAGE in ("de", "german"):
654         df = df[df["review_text"].apply(is_german_review)].copy()
655     elif LANGUAGE in ("en", "english"):
656         df = df[df["review_text"].apply(is_english_review)].copy()
657     # else auto: keep all
```

```
658
659     print(f" Reviews after filtering: {len(df)}")
660
661     if MAX_REVIEWS:
662         df = df.head(MAX_REVIEWS).copy()
663         print(f" Processing subset: {MAX_REVIEWS} rows.")
664
665     # 3) ANALYSIS LOOP
666     print("\n[2] Extracting Facts (Strict Mode + LLM)...")
667     results = []
668
669     cache_path = run_dir / CACHE_FILE_GENERIC
670     cache_bk_path = run_dir / CACHE_FILE_BK
671
672     if cache_path.exists():
673         try:
674             results = pd.read_csv(cache_path).to_dict("records")
675             print(f" Resumed {len(results)} from cache.")
676         except Exception:
677             pass
678
679     processed_ids = set([str(r.get("idx")) for r in results])
680
681     for idx, row in tqdm(df.iterrows(), total=len(df)):
682         idx_str = str(idx)
683         if idx_str in processed_ids:
684             continue
685
686         text = row["review_text"]
687         stars = row.get("stars", pd.NA)
688         date_val = row.get("parsed_date", pd.NaT)
689
690         json_k = parse_json_response(call_llm(get_kitchen_prompt(text, stars)))
691         json_s = parse_json_response(call_llm(get_service_prompt(text, stars)))
692         json_m = parse_json_response(call_llm(get_mgmt_prompt(text, stars)))
693         json_sem = parse_json_response(call_llm(get_sentiment_prompt(text, stars)
                ))
694
695         vscore = get_vader_score(text)
696
697         llm_label_raw = json_sem.get("label", json_sem.get("sentiment", "neutral"
                ))
698         llm_conf = json_sem.get("confidence", 0.6)
699         llm_label_norm = normalize_llm_label(llm_label_raw)
700         llm_score = llm_label_to_score(llm_label_norm, llm_conf)
701         fused_label, fused_score, w_v, w_l = fuse_sentiment(vscore,
                llm_label_norm, llm_conf)
702
703         results.append({
704             "idx": idx_str,
705             "date": date_val,
```

```
            "stars": stars,
            "review_text": text,
            "vader_score": vscore,
            "sentiment_bucket": vader_bucket(vscore),
            "sentiment_label": llm_label_norm.title(), # legacy-friendly
            "llm_label": llm_label_norm,
            "llm_confidence": llm_conf,
            "llm_score": llm_score,
            "fused_label": fused_label,
            "fused_score": fused_score,
            "fused_w_vader": w_v,
            "fused_w_llm": w_l,
            "kitchen_pros": json_k.get("kitchen_pros", "n/a"),
            "kitchen_cons": json_k.get("kitchen_cons", "n/a"),
            "service_pros": json_s.get("service_pros", "n/a"),
            "service_cons": json_s.get("service_cons", "n/a"),
            "mgmt_pros": json_m.get("mgmt_pros", "n/a"),
            "mgmt_cons": json_m.get("mgmt_cons", "n/a"),
        })

        # save cache often
        if len(results) % 5 == 0:
            pd.DataFrame(results).to_csv(cache_path, index=False)
            if WRITE_LEGACY_OUTPUTS:
                pd.DataFrame(results).to_csv(cache_bk_path, index=False)

        time.sleep(0.02)

    final_df = pd.DataFrame(results)

    # cleanup: remove n/a
    for col in final_df.columns:
        if "pros" in col or "cons" in col:
            final_df[col] = final_df[col].replace(["n/a", "N/A", "nichts", "keine
                "], "")

    # --- FIX: ensure stars are numeric 1..5 for distribution/chart ---
    final_df["stars"] = final_df["stars"].apply(clamp_star)

    # Save processed outputs (canonical; optional legacy)
    processed_path = run_dir / PROCESSED_FILE_GENERIC
    processed_bk_path = run_dir / PROCESSED_FILE_BK

    final_df.to_csv(processed_path, index=False)
    if WRITE_LEGACY_OUTPUTS:
        final_df.to_csv(processed_bk_path, index=False)

    if WRITE_LEGACY_OUTPUTS:
        print(f"\n[3] Data saved to {processed_path.name} (+ legacy {
            processed_bk_path.name})")
    else:
```

```
        print(f"\n[3] Data saved to {processed_path.name}")

    if final_df.empty:
        print("No data to summarize.")
        return

    # 4) OWNER SUMMARY JSON
    print("\n" + "=" * 60)
    print("OWNER SUMMARY (LLM - ALL REVIEWS)")
    print("=" * 60)

    evidence_lines = build_evidence_lines(final_df)

    chunk_outputs = []
    for chunk in chunk_list(evidence_lines, chunk_size=10):
        prompt = chunk_summary_prompt(chunk)
        resp = parse_json_response(call_llm(prompt, timeout=180, retries=3))
        if isinstance(resp, dict) and resp:
            chunk_outputs.append(resp)

    merged = merge_chunk_outputs(chunk_outputs)
    final_prompt = final_owner_summary_prompt(merged)
    owner_summary = parse_json_response(call_llm(final_prompt, timeout=180,
        retries=3))
    if not isinstance(owner_summary, dict):
        owner_summary = {}

    owner_json_path = run_dir / OWNER_JSON_GENERIC
    owner_json_bk_path = run_dir / OWNER_JSON_BK
    with open(owner_json_path, "w", encoding="utf-8") as f:
        json.dump(owner_summary, f, ensure_ascii=False, indent=2)
    if WRITE_LEGACY_OUTPUTS:
        with open(owner_json_bk_path, "w", encoding="utf-8") as f:
            json.dump(owner_summary, f, ensure_ascii=False, indent=2)

    # write txt (readable later by owner_outputs.py too)
    owner_txt_path = run_dir / OWNER_TXT_GENERIC
    owner_txt_bk_path = run_dir / OWNER_TXT_BK
    owner_txt_path.write_text(json.dumps(owner_summary, ensure_ascii=False,
        indent=2), encoding="utf-8")
    if WRITE_LEGACY_OUTPUTS:
        owner_txt_bk_path.write_text(json.dumps(owner_summary, ensure_ascii=False
            , indent=2), encoding="utf-8")

    print(f"Saved owner summary: {owner_json_path.name}" + (f" (and {
        owner_json_bk_path.name})" if WRITE_LEGACY_OUTPUTS else ""))

    # 5) CHARTS
    print("\n[4] Generating charts...")

    # Chart: star distribution (use numeric stars only)
```

```
    stars_series = pd.to_numeric(final_df["stars"], errors="coerce").dropna().
        astype(int)
    plt.figure(figsize=(8, 5))
    stars_series.value_counts().sort_index().reindex([1,2,3,4,5], fill_value=0).
        plot(kind="bar", edgecolor="black")
    plt.title("Star Rating Distribution", fontsize=14)
    plt.xlabel("Stars")
    plt.ylabel("Count")
    plt.xticks(rotation=0)
    plt.tight_layout()
    plt.savefig(run_dir / CHART_STARS)
    if WRITE_LEGACY_OUTPUTS:
        plt.savefig(run_dir / CHART1_BK)
    plt.close()

    # Chart: sentiment pie
    plt.figure(figsize=(6, 6))
    label_col = "fused_label" if "fused_label" in final_df.columns else "
        sentiment_bucket"
    counts = final_df[label_col].value_counts()
    plt.pie(counts, labels=counts.index, autopct="%1.1f%%")
    plt.title(f"Overall Sentiment ({'Fused' if label_col=='fused_label' else '
        VADER Buckets'})", fontsize=14)
    plt.ylabel("")
    plt.tight_layout()
    plt.savefig(run_dir / CHART_SENTIMENT)
    if WRITE_LEGACY_OUTPUTS:
        plt.savefig(run_dir / CHART2_BK)
    plt.close()

    # Chart: trend
    trend_df = final_df.copy()
    trend_df["date"] = pd.to_datetime(trend_df["date"], errors="coerce")
    trend_df = trend_df.dropna(subset=["date"]).sort_values("date")

    if not trend_df.empty:
        trend_df = trend_df.set_index("date")
        score_col = "fused_score" if "fused_score" in trend_df.columns else "
            vader_score"
        monthly_trend = trend_df[score_col].resample("ME").mean()

        plt.figure(figsize=(10, 5))
        plt.plot(monthly_trend.index, monthly_trend.values, marker="o", linestyle
            ="-", linewidth=2)
        plt.axhline(0, linestyle="--")
        plt.title(f"Customer Satisfaction Trend (Monthly Average - {'Fused' if
            score_col=='fused_score' else 'VADER'})", fontsize=14)
        plt.ylabel("Sentiment Score (-1.0 to +1.0)")
        plt.xlabel("Date")
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
```

```
846         plt.savefig(run_dir / CHART_TREND)
847         if WRITE_LEGACY_OUTPUTS:
848             plt.savefig(run_dir / CHART3_BK)
849         plt.close()
850 
851 
852 
853     # ----------------------------
854     # TERMINAL SUMMARY (quick KPIs)
855     # ----------------------------
856     try:
857         print("\n========== RUN SUMMARY ==========")
858         total = len(final_df)
859         print(f"Total reviews processed: {total}")
860 
861         # Stars
862         if "stars" in final_df.columns:
863             stars_series = pd.to_numeric(final_df["stars"], errors="coerce").
                dropna()
864             if len(stars_series) > 0:
865                 print(f"Average star rating: {stars_series.mean():.2f} / 5")
866                 dist = stars_series.value_counts().sort_index()
867                 print("Stars distribution:")
868                 for s, c in dist.items():
869                     try:
870                         s_int = int(float(s))
871                     except Exception:
872                         s_int = s
873                     print(f" {s_int}* : {int(c)}")
874 
875         # Sentiment bucket (VADER)
876         if "sentiment_bucket" in final_df.columns:
877             counts = final_df["sentiment_bucket"].fillna("Unknown").value_counts
                ()
878             print("Sentiment distribution (VADER):")
879             for k, v in counts.items():
880                 print(f" {k}: {int(v)}")
881 
882         # Fused sentiment (recommended)
883         if "fused_label" in final_df.columns:
884             fused_counts = final_df["fused_label"].fillna("Neutral").value_counts
                ()
885             print("Sentiment distribution (Fused):")
886             for k, v in fused_counts.items():
887                 print(f" {k}: {int(v)}")
888 
889             # Disagreement rates (useful sanity check)
890             try:
891                 llm_title = final_df["llm_label"].fillna("neutral").apply(
                    normalize_llm_label).str.title()
892                 vader_title = final_df["sentiment_bucket"].fillna("Neutral")
```

```
                disagree_llm_vader = (llm_title != vader_title).mean() * 100.0
                disagree_fused_vader = (final_df["fused_label"].fillna("Neutral")
                    != vader_title).mean() * 100.0
                print(f"Disagreement (LLM vs VADER): {disagree_llm_vader:.1f}%")
                print(f"Disagreement (Fused vs VADER): {disagree_fused_vader:.1f}%
                    ")
            except Exception:
                pass


        # Sentiment label from LLM (optional)
        if "llm_label" in final_df.columns:
            counts2 = final_df["llm_label"].fillna("neutral").apply(
                normalize_llm_label).value_counts()
            # show top 5 to avoid spam
            print("Sentiment labels (LLM) -top 5 (normalized):")
            for k, v in counts2.head(5).items():
                print(f" {k}: {int(v)}")

        print("================================\n")
    except Exception as e:
        print("[WARN] Could not print run summary:", e)


    print("\nDone. Run folder:")
    print(f"- {run_dir}")


def run_subprocess_step(script_name: str):
    """Run a helper script without killing the whole pipeline."""
    import sys
    import subprocess

    script_path = HERE / script_name
    if not script_path.exists():
        print(f"[WARN] Missing script: {script_name} (skipping)")
        return

    print(f"\n[STEP] Running: {script_name}")
    result = subprocess.run(
        [sys.executable, str(script_path)],
        cwd=str(HERE),
        env=os.environ.copy(),
        check=False
    )
    if result.returncode == 0:
        print(f"[OK] {script_name} finished successfully.")
    else:
        print(f"[WARN] {script_name} finished with return code {result.returncode
            }.")

```

```
940
941 if __name__ == "__main__":
942     main()
943     run_subprocess_step("owner_outputs.py")
944     run_subprocess_step("send_weekly_report.py")
```

F.2 Owner outputs: `owner_outputs.py`

F.2.1 Description

This script turns run artifacts into owner-facing outputs:

- loads `owner_summary.json` and creates a readable text version,
- produces `owner_summary_flat.csv` for BI tools,
- generates `weekly_owner_email.txt` which is later sent by the email script.

F.2.2 Code

Listing F.2: Owner output generation

```
1  import json
2  import os
3  import re
4  from datetime import datetime
5  from pathlib import Path
6  from collections import Counter
7  import pandas as pd
8
9  BASE_DIR = Path(__file__).resolve().parent
10
11 RESTAURANT_NAME = os.getenv("RESTAURANT_NAME", "").strip() or "Restaurant"
12 MODEL_NAME = os.getenv("MODEL_NAME", "").strip() or "LLM"
13 WRITE_LEGACY_OUTPUTS = os.getenv("WRITE_LEGACY_OUTPUTS", "0").strip().lower() in
       ("1", "true", "yes", "y")
14
15 RUNS_ROOT = BASE_DIR / "runs"
16
17
18 def clean_text(s: str) -> str:
19     s = "" if s is None else str(s)
20     s = re.sub(r"\s+", " ", s).strip()
21     return s
22
23
24 def as_list(x):
25     if x is None:
26         return []
27     if isinstance(x, list):
28         return x
```

```
    if isinstance(x, str):
        t = x.strip()
        if not t:
            return []
        return [t]
    return [x]


def normalize_area(s: str) -> str:
    s = clean_text(s)
    if not s:
        return "General"
    # Keep short, avoid multi slashes explosion
    s = s.replace("\\", "/")
    s = re.sub(r"\s*/\s*", "/", s)
    return s


def get_run_dir() -> Path:
    env_dir = os.getenv("RUN_DIR", "").strip()
    if env_dir:
        d = Path(env_dir)
        d.mkdir(parents=True, exist_ok=True)
        return d

    last = RUNS_ROOT / "last_run.txt"
    if last.exists():
        try:
            d = Path(last.read_text(encoding="utf-8").strip())
            if d.exists():
                os.environ["RUN_DIR"] = str(d)
                return d
        except Exception:
            pass

    # fallback: newest run folder
    RUNS_ROOT.mkdir(exist_ok=True)
    runs = sorted([p for p in RUNS_ROOT.iterdir() if p.is_dir()], key=lambda p:
        p.stat().st_mtime, reverse=True)
    if runs:
        os.environ["RUN_DIR"] = str(runs[0])
        return runs[0]

    # last fallback: create one
    d = RUNS_ROOT / datetime.now().strftime("%Y-%m-%d_%H-%M")
    d.mkdir(parents=True, exist_ok=True)
    os.environ["RUN_DIR"] = str(d)
    return d


def load_owner_summary(run_dir: Path) -> dict:
```

```
    # Prefer canonical names; fall back to legacy
    cand = [
        run_dir / "owner_summary.json",
        run_dir / "bk_owner_summary.json",
        BASE_DIR / "owner_summary.json",
        BASE_DIR / "bk_owner_summary.json",
    ]
    for p in cand:
        if p.exists():
            try:
                return json.loads(p.read_text(encoding="utf-8"))
            except Exception:
                pass
    return {}


def load_processed_reviews(run_dir: Path) -> pd.DataFrame:
    cand = [
        run_dir / "reviews_processed.csv",
        run_dir / "bk_final_report.csv",
        BASE_DIR / "reviews_processed.csv",
        BASE_DIR / "bk_final_report.csv",
    ]
    for p in cand:
        if p.exists():
            try:
                df = pd.read_csv(p)
                return df
            except Exception:
                pass
    return pd.DataFrame()


def split_phrases(cell) -> list:
    """
    Extract short phrases from pros/cons cells.
    Handles:
      - NaN
      - single sentence
      - semicolon separated phrases
    """
    if cell is None:
        return []
    s = str(cell).strip()
    if not s or s.lower() == "nan":
        return []
    # split by ; or | first, then by newlines
    parts = re.split(r"[;\n\|]+", s)
    out = []
    for p in parts:
        t = clean_text(p)
```

```
        # keep reasonably short phrases
        if 2 <= len(t) <= 140:
            out.append(t)
    return out


def top_phrases(df: pd.DataFrame, positive: bool = True, k: int = 3):
    if df.empty:
        return []

    # determine label column preference
    label_col = None
    if "fused_label" in df.columns:
        label_col = "fused_label"
        pos_key, neg_key = "Positive", "Negative"
    elif "sentiment_bucket" in df.columns:
        label_col = "sentiment_bucket"
        pos_key, neg_key = "Positive", "Negative"
    elif "sentiment_label" in df.columns:
        label_col = "sentiment_label"
        pos_key, neg_key = "Positive", "Negative"

    if not label_col:
        return []

    target = pos_key if positive else neg_key
    sub = df[df[label_col].astype(str).str.lower() == target.lower()].copy()
    if sub.empty:
        return []

    cols = ["kitchen_pros", "service_pros", "mgmt_pros"] if positive else ["
        kitchen_cons", "service_cons", "mgmt_cons"]
    existing = [c for c in cols if c in sub.columns]
    if not existing:
        return []

    cnt = Counter()
    for c in existing:
        for cell in sub[c].tolist():
            for phrase in split_phrases(cell):
                cnt[phrase] += 1

    return cnt.most_common(k)


def compute_snapshot(df: pd.DataFrame) -> dict:
    snap = {
        "n_reviews": int(len(df)) if df is not None else 0,
        "avg_stars": None,
        "star_counts": {},
        "sent_counts": {},
```

```
        "overall_score": None,
        "score_source": None,
        "label_source": None,
    }
    if df is None or df.empty:
        return snap

    # stars
    if "stars" in df.columns:
        stars = pd.to_numeric(df["stars"], errors="coerce")
        stars = stars.dropna()
        if not stars.empty:
            snap["avg_stars"] = float(stars.mean())
            vc = stars.round().astype(int).value_counts().to_dict()
            snap["star_counts"] = {f"{i}*": int(vc.get(i, 0)) for i in [1, 2, 3,
                4, 5]}

    # sentiment (prefer fused)
    if "fused_label" in df.columns:
        lab = df["fused_label"].astype(str)
        vc = lab.value_counts().to_dict()
        snap["sent_counts"] = {k: int(v) for k, v in vc.items()}
        snap["label_source"] = "Fused"
    elif "sentiment_bucket" in df.columns:
        lab = df["sentiment_bucket"].astype(str)
        vc = lab.value_counts().to_dict()
        snap["sent_counts"] = {k: int(v) for k, v in vc.items()}
        snap["label_source"] = "VADER"

    # overall score (prefer fused_score, else vader_score)
    if "fused_score" in df.columns:
        sc = pd.to_numeric(df["fused_score"], errors="coerce").dropna()
        if not sc.empty:
            snap["overall_score"] = float(sc.mean())
            snap["score_source"] = "Fused"
    elif "vader_score" in df.columns:
        sc = pd.to_numeric(df["vader_score"], errors="coerce").dropna()
        if not sc.empty:
            snap["overall_score"] = float(sc.mean())
            snap["score_source"] = "VADER"

    return snap


def build_email_text(summary: dict, snapshot: dict, top_pos, top_neg) -> str:
    """
    Keep the *previous version* structure, but show fused metrics when available
        .
    """
    now = datetime.now().strftime("%Y-%m-%d %H:%M")
    run_date = os.getenv("RUN_STARTED_AT", "") or now
```

```
229
230     # subject line stays exactly like previous version
231     subject = f"Subject: Weekly Review Summary -{RESTAURANT_NAME} -{datetime.now
            ().strftime('%Y-%m-%d')}"
232
233     headline = clean_text(summary.get("headline", "Restaurant Performance
            Enhancement Guide for Owner"))
234     lines = []
235     lines.append(subject)
236     lines.append("")
237     lines.append(f"{headline} -{RESTAURANT_NAME}")
238     lines.append(f"Model: {MODEL_NAME}")
239     lines.append(f"Run date: {run_date}")
240     lines.append("")
241
242     lines.append("DATA SNAPSHOT")
243     lines.append(f"• Reviews analyzed: {snapshot.get('n_reviews', 0)}")
244
245     score = snapshot.get("overall_score", None)
246     score_src = snapshot.get("score_source", "")
247     if score is None:
248         lines.append("• Overall sentiment: n/a")
249     else:
250         # show source explicitly, but keep old formatting
251         tag = f"{score_src} avg" if score_src else "avg"
252         lines.append(f"• Overall sentiment ({tag}): {score:.2f} (-1 to +1)")
253
254     # sentiment counts (prefer fused)
255     sent = snapshot.get("sent_counts", {}) or {}
256     label_src = snapshot.get("label_source", "")
257     if sent:
258         # normalize keys to Positive/Neutral/Negative order
259         def pick(d, key):
260             for k in d.keys():
261                 if str(k).lower() == key.lower():
262                     return int(d[k])
263             return 0
264         p = pick(sent, "Positive")
265         n = pick(sent, "Negative")
266         u = pick(sent, "Neutral")
267         tag = label_src if label_src else "Sentiment"
268         lines.append(f"• Sentiment ({tag}): Positive={p}, Neutral={u}, Negative={
                n}")
269
270     stars = snapshot.get("star_counts", {}) or {}
271     if stars:
272         star_str = ", ".join([f"{k}={stars.get(k,0)}" for k in ["1*", "2*", "3*",
                 "4*", "5*"]])
273         lines.append(f"• Stars: {star_str}")
274     lines.append("")
275
```

```
    # strengths
    lines.append("TOP STRENGTHS (keep & promote)")
    exc = as_list(summary.get("areas_of_excellence", []))[:3]
    if not exc:
        lines.append("• No strong repeated strengths detected in this run.")
    else:
        for i, it in enumerate(exc, 1):
            area = normalize_area(it.get("area", ""))
            theme = clean_text(it.get("theme", ""))
            keep = clean_text(it.get("what_to_keep", "")) or "Beibehalten und
                standardisieren."
            promo = clean_text(it.get("how_to_market_it", "")) or "In Google/
                Online-Bewertungen hervorheben."
            ev = [clean_text(x) for x in as_list(it.get("evidence", [])) if
                clean_text(x)]
            lines.append(f"{i}. [{area}] {theme}")
            lines.append(f" Keep: {keep}")
            lines.append(f" Promote: {promo}")
            if ev:
                lines.append(f' Evidence: "{ev[0]}"')
    lines.append("")

    # issues
    lines.append("TOP ISSUES (prioritized)")
    imp = as_list(summary.get("areas_of_improvement", []))[:5]
    if not imp:
        lines.append("• No major repeated issues detected in this run.")
    else:
        for i, it in enumerate(imp, 1):
            area = normalize_area(it.get("area", ""))
            theme = clean_text(it.get("theme", ""))
            fix = clean_text(it.get("what_to_fix", "")) or "Problem genauer
                beschreiben und messen."
            action = clean_text(it.get("suggested_action", "")) or "Klaren
                Standardprozess definieren und schulen."
            ev = [clean_text(x) for x in as_list(it.get("evidence", [])) if
                clean_text(x)]
            lines.append(f"{i}. [{area}] {theme}")
            lines.append(f" Fix: {fix}")
            lines.append(f" Action: {action}")
            if ev:
                lines.append(f' Evidence: "{ev[0]}"')
    lines.append("")

    # quotes
    lines.append("RECURRING QUOTES (from extraction)")
    if top_pos:
        lines.append("• Positive:")
        for phrase, cnt in top_pos[:3]:
            lines.append(f' - "{phrase}" ({cnt}x)')
    if top_neg:
```

```
321         lines.append("• Negative:")
322         for phrase, cnt in top_neg[:3]:
323             lines.append(f' - "{phrase}" ({cnt}x)')
324     if not top_pos and not top_neg:
325         lines.append("• No recurring phrases extracted in this run.")
326     lines.append("")
327 
328     # action plan (support both schemas; fill from improvements if missing)
329     lines.append("ACTION PLAN")
330 
331     quick = as_list(summary.get("quick_wins_next_7_days", []))[:3]
332     longer = as_list(summary.get("longer_term_30_days", []))[:3]
333 
334     # If missing, derive from improvements suggested actions
335     if not quick and imp:
336         quick = [clean_text(x.get("suggested_action", "")) for x in imp[:3] if
                clean_text(x.get("suggested_action", ""))]
337         quick = quick[:3]
338     if not longer and imp:
339         longer = [clean_text(x.get("suggested_action", "")) for x in imp[3:8] if
                clean_text(x.get("suggested_action", ""))]
340         longer = longer[:3]
341 
342     lines.append("• Quick wins (next 7 days):")
343     if quick:
344         for x in quick:
345             lines.append(f" - {clean_text(x)}")
346     else:
347         lines.append(" - Standardize one service/kitchen checklist for every
                shift.")
348         lines.append(" - Review top 5 negative reviews with staff and agree on
                fixes.")
349 
350     lines.append("• Longer term (next 30 days):")
351     if longer:
352         for x in longer:
353             lines.append(f" - {clean_text(x)}")
354     else:
355         lines.append(" - Reduce repeated complaints by adding process + training
                + owner checks.")
356 
357     lines.append("")
358 
359     # KPIs (support both schemas; fall back to sensible defaults)
360     kpis = as_list(summary.get("suggested_kpis", []))[:4]
361     if not kpis:
362         # Some older schemas used different key names
363         kpis = as_list(summary.get("suggested_kpi_for_30_days", []))[:4]
364 
365     lines.append("KPIs TO TRACK (simple)")
366     if kpis:
```

```
        for x in kpis[:4]:
            lines.append(f"• {clean_text(x)}")
    else:
        lines.append("• Average Wait Time (Min.) -Timer: 20 Bestellungen pro Schicht messen.")
        lines.append("• Order Accuracy (%) -Stichprobe: 30 Bestellungen/Woche prüfen.")
        lines.append("• Food Quality Rating (1-5) -QR-Umfrage auf Bon / Tischaufsteller.")
        lines.append("• Cleanliness Checklist Score -Tägliche Checkliste + wöchentliches Audit.")

    lines.append("")
    lines.append(f"Generated on: {now}")
    return "\n".join(lines)


def write_flat_csv(summary: dict, run_dir: Path):
    now = datetime.now().strftime("%Y-%m-%d %H:%M")
    rows = []
    for it in as_list(summary.get("areas_of_excellence", [])):
        rows.append({
            "generated_at": now,
            "category": "excellence",
            "area": normalize_area(it.get("area", "")),
            "theme": clean_text(it.get("theme", "")),
            "keep_or_fix": clean_text(it.get("what_to_keep", "")),
            "action_or_marketing": clean_text(it.get("how_to_market_it", "")),
        })
    for it in as_list(summary.get("areas_of_improvement", [])):
        rows.append({
            "generated_at": now,
            "category": "improvement",
            "area": normalize_area(it.get("area", "")),
            "theme": clean_text(it.get("theme", "")),
            "keep_or_fix": clean_text(it.get("what_to_fix", "")),
            "action_or_marketing": clean_text(it.get("suggested_action", "")),
        })
    if not rows:
        return

    out = run_dir / "owner_summary_flat.csv"
    pd.DataFrame(rows).to_csv(out, index=False, encoding="utf-8")
    if WRITE_LEGACY_OUTPUTS:
        pd.DataFrame(rows).to_csv(run_dir / "bk_owner_summary_flat.csv", index=False, encoding="utf-8")


def main():
    run_dir = get_run_dir()
    summary = load_owner_summary(run_dir)
```

```
    df = load_processed_reviews(run_dir)

    # snapshot + quotes
    snapshot = compute_snapshot(df)
    top_pos = top_phrases(df, positive=True, k=3)
    top_neg = top_phrases(df, positive=False, k=3)

    # write owner_summary_readable + weekly email
    email_text = build_email_text(summary, snapshot, top_pos, top_neg)

    readable = run_dir / "owner_summary_readable.txt"
    readable.write_text(email_text, encoding="utf-8")
    if WRITE_LEGACY_OUTPUTS:
        (run_dir / "bk_owner_summary.txt").write_text(email_text, encoding="utf-8
            ")

    weekly = run_dir / "weekly_owner_email.txt"
    weekly.write_text(email_text, encoding="utf-8")

    # also copy JSON to run folder (if summary came from base dir)
    out_json = run_dir / "owner_summary.json"
    if not out_json.exists() and summary:
        out_json.write_text(json.dumps(summary, ensure_ascii=False, indent=2),
            encoding="utf-8")
    if WRITE_LEGACY_OUTPUTS and summary:
        (run_dir / "bk_owner_summary.json").write_text(json.dumps(summary,
            ensure_ascii=False, indent=2), encoding="utf-8")

    write_flat_csv(summary, run_dir)

    print("OWNER OUTPUTS")
    print("=" * 40)
    print(f"Run dir: {run_dir}")
    print(f"Wrote: {weekly.name}, {readable.name}, owner_summary_flat.csv")


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

F.3 Email sending: `send_weekly_report.py`

F.3.1 Description

This script sends the weekly owner email with attachments:

- reloads environment variables from `.env`,
- resolves the run folder (`RUN_DIR` or `runs/last_run.txt`),
- loads subject/body from `weekly_owner_email.txt`,
- attaches charts and summary files present in the run folder,

- sends the email via SMTP.

F.3.2 Code

Listing F.3: Weekly email sending script

```
import os
from datetime import datetime
from pathlib import Path
from email_reporter import send_email_with_attachments

BASE_DIR = Path(__file__).resolve().parent
RUNS_ROOT = BASE_DIR / "runs"


def load_dotenv(path: Path, overwrite: bool = True):
    """Minimal .env loader (overwrite=True avoids old env values)."""
    if not path.exists():
        print(f"[ENV] .env not found at: {path}")
        return

    print(f"[ENV] Loading .env from: {path}")
    for line in path.read_text(encoding="utf-8").splitlines():
        line = line.strip()
        if not line or line.startswith("#") or "=" not in line:
            continue
        k, v = line.split("=", 1)
        k = k.strip()
        v = v.strip().strip('"').strip("'")
        if not k:
            continue
        if overwrite or (k not in os.environ):
            os.environ[k] = v


def resolve_run_dir() -> Path:
    env_dir = os.getenv("RUN_DIR", "").strip()
    if env_dir:
        d = Path(env_dir)
        d.mkdir(parents=True, exist_ok=True)
        return d

    last = RUNS_ROOT / "last_run.txt"
    if last.exists():
        try:
            d = Path(last.read_text(encoding="utf-8").strip())
            if d.exists():
                os.environ["RUN_DIR"] = str(d)
                return d
        except Exception:
            pass
```

```
46
47      return BASE_DIR
48
49
50  def load_subject_and_body(path: Path):
51      content = path.read_text(encoding="utf-8").strip()
52      subject = "Weekly Restaurant Review Report"
53      body = content
54
55      if content.lower().startswith("subject:"):
56          lines = content.splitlines()
57          subject = lines[0].split(":", 1)[1].strip()
58          body = "\n".join(lines[1:]).strip()
59
60      return subject, body
61
62
63  def parse_recipients(value: str):
64      if not value:
65          return []
66      parts = [p.strip() for p in value.replace(";", ",").split(",")]
67      return [p for p in parts if p]
68
69
70  def main():
71      print("=" * 60)
72      print("SENDING WEEKLY OWNER REPORT (Email)")
73      print("=" * 60)
74
75      # Always reload env
76      load_dotenv(BASE_DIR / ".env", overwrite=True)
77
78      run_dir = resolve_run_dir()
79      print(f"[RUN FOLDER] {run_dir}")
80
81      email_text_file = run_dir / "weekly_owner_email.txt"
82      if not email_text_file.exists():
83          raise SystemExit(f"Missing email text file: {email_text_file} (run
                owner_outputs.py first)")
84
85      smtp_host = os.getenv("SMTP_HOST", "").strip()
86      smtp_port = int((os.getenv("SMTP_PORT", "587").strip() or "587"))
87      smtp_user = os.getenv("SMTP_USER", "").strip()
88      smtp_pass = os.getenv("SMTP_PASS", "").strip()
89
90      recipients = parse_recipients(os.getenv("OWNER_EMAILS", "").strip())
91      if not recipients:
92          recipients = parse_recipients(os.getenv("OWNER_EMAIL", "").strip())
93
94      missing = [k for k in ["SMTP_HOST", "SMTP_USER", "SMTP_PASS"] if not os.
            getenv(k)]
```

```
95      if missing:
96          raise SystemExit("Missing env vars: " + ", ".join(missing))
97      if not recipients:
98          raise SystemExit("Missing recipient. Set OWNER_EMAIL (or OWNER_EMAILS) in
                .env")
99
100     subject, body = load_subject_and_body(email_text_file)
101
102     # ensure date visible even if subject line didn't include it
103     if "-" not in subject:
104         subject = f"{subject} -{datetime.now().strftime('%Y-%m-%d')}"
105
106     # Attach whatever exists (generic + backward compatible)
107     attachment_names = [
108         # charts
109         "chart_stars.png",
110         "chart_sentiment_pie.png",
111         "chart_trend.png",
112         # summaries
113         "owner_summary.json",
114         "owner_summary_readable.txt",
115         "owner_summary_flat.csv",
116         "weekly_owner_email.txt",
117         # processed data
118         "reviews_processed.csv",
119         "cache_reviews.csv",
120     ]
121
122     attachments = []
123     for name in attachment_names:
124         p = run_dir / name
125         if p.exists():
126             attachments.append(str(p))
127
128     print("\n[ENV CHECK]")
129     print(" SMTP_HOST:", smtp_host)
130     print(" SMTP_PORT:", smtp_port)
131     print(" SMTP_USER:", smtp_user)
132     print(" RECIPIENTS:", ", ".join(recipients))
133
134     print("\n[ATTACHMENTS]")
135     if attachments:
136         for a in attachments:
137             print(" +", a)
138     else:
139         print(" (No attachments found!)")
140
141     ok_all = True
142     for to_email in recipients:
143         print(f"\n[SENDING EMAIL] -> {to_email}")
144         ok = send_email_with_attachments(
```

```
            to_email=to_email,
            subject=subject,
            body=body,
            attachments=attachments,
            smtp_host=smtp_host,
            smtp_port=smtp_port,
            smtp_user=smtp_user,
            smtp_pass=smtp_pass,
        )
        print(" RESULT:", "[OK] Sent!" if ok else "[X] Failed")
        ok_all = ok_all and ok

    return 0 if ok_all else 1


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

F.4 SMTP helper: email_reporter.py

F.4.1 Description

This module encapsulates the low-level SMTP logic to build and send an email with attachments using Python's standard library.

F.4.2 Code

Listing F.4: SMTP email helper

```
import os
import mimetypes
import smtplib
import ssl
from pathlib import Path
from email.message import EmailMessage


def _dedupe_attachments(attachments: list) -> list:
    """Remove duplicate attachments safely.

    - Dedupes by resolved path first.
    - If two different files share a name, keeps both (email clients show names)
      ,
      but you can still avoid accidental duplicates by passing a clean list.
    """
    seen = set()
    out = []
    for a in attachments or []:
        if a is None:
            continue
```

```
        p = Path(a).expanduser()
        try:
            key = str(p.resolve())
        except Exception:
            key = str(p)
        if key in seen:
            continue
        seen.add(key)
        out.append(p)
    return out


def send_email_with_attachments(
    to_email: str,
    subject: str,
    body: str,
    attachments: list,
    smtp_host: str,
    smtp_port: int,
    smtp_user: str,
    smtp_pass: str,
) -> bool:
    """Send an email with optional attachments.

    attachments: list of file paths (str/Path). Missing files are skipped.
    """
    try:
        msg = EmailMessage()
        msg["From"] = smtp_user
        msg["To"] = to_email
        msg["Subject"] = subject
        msg.set_content(body or "")

        # Attach files (deduped + skip missing)
        for path in _dedupe_attachments(attachments):
            if not path.exists() or not path.is_file():
                print(f" [SKIP] Attachment not found: {path}")
                continue

            ctype, encoding = mimetypes.guess_type(str(path))
            if ctype is None or encoding is not None:
                ctype = "application/octet-stream"
            maintype, subtype = ctype.split("/", 1)

            with open(path, "rb") as f:
                data = f.read()

            msg.add_attachment(
                data,
                maintype=maintype,
                subtype=subtype,
```

```
72                filename=path.name,
73            )
74
75        context = ssl.create_default_context()
76
77        # 465 = implicit TLS, 587 = STARTTLS
78        if int(smtp_port) == 465:
79            with smtplib.SMTP_SSL(smtp_host, smtp_port, context=context) as
                  server:
80                server.login(smtp_user, smtp_pass)
81                server.send_message(msg)
82        else:
83            with smtplib.SMTP(smtp_host, smtp_port) as server:
84                server.ehlo()
85                server.starttls(context=context)
86                server.ehlo()
87                server.login(smtp_user, smtp_pass)
88                server.send_message(msg)
89
90        return True
91
92    except Exception as e:
93        print("Email send failed:", e)
94        return False
```

Appendix G

Bill of Material (software)

Component	Purpose
Python 3.10+	Runtime
pandas	CSV handling and transformations
NLTK (VADER)	Baseline sentiment scoring
matplotlib	Chart generation
requests	HTTP communication with the local LLM endpoint
Ollama	Local LLM runtime providing an API (`/api/generate`)
phi3.5 (LLM model)	Model used via Ollama for evidence extraction and sentiment labeling
SMTP (Python stdlib)	Email reporting and delivery

The `requirements.txt` file in the repository lists the Python dependencies required to run the pipeline.

Bibliography

[1] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. O'Reilly Media, Inc., 2009, ISBN: 978-0-596-51649-9.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, 2019.

[3] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *AI Magazine*, vol. 17, no. 3, pp. 37–54, 1996.

[4] C. J. Hutto and E. Gilbert, "Vader: A parsimonious rule-based model for sentiment analysis of social media text," in *Proceedings of the International AAAI Conference on Web and Social Media*, vol. 8, 2014, pp. 216–225. DOI: 10.1609/icwsm.v8i1.14550.

[5] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference (SciPy)*, 2010.

[6] L. Ouyang et al., "Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[7] pandas Development Team, *Pandas: Citing and logo*, Accessed: 2026-01-07, 2025. [Online]. Available: https://pandas.pydata.org/about/citing.html.