listingname=Source Code

Computer Science Project
Kandy: CLI tool for managing Kafka

Alex Lukin
Chesterton Community College
Cambridge, United Kingdom

December 9, 2024

Contents

1 Analysis

1.1 Problem Definition

Apache Kafka[1] has become a crucial system for handling real-time data streams in today's data-driven landscape. Despite Kafka's widespread adoption, there remains a lack of streamlined tools for efficient management of Kafka instances directly from the command line. This gap poses a challenge for developers who need quick, straightforward ways to interact with Kafka without getting slowed down by complex setup processes.

While web applications can bridge this gap by providing intuitive interfaces, they are typically hosted in Docker containers. Docker's benefits of portability and consistency across different environments are valuable, yet setting up these applications within large systems can become cumbersome and time-consuming, especially for quick local tests.

In summary, Docker facilitates deploying web applications with user-friendly interfaces for Kafka management, but its setup can be daunting within larger infrastructures. To address this, the proposed solution is a dedicated Text-Based User Interface (TUI) utility aimed at simplifying Kafka management, ultimately boosting developer productivity.

The project is named **Kandy**(Stands for Handy Kafka) and is envisioned as a TUI tool specifically designed for Kafka management

1.2 Stakeholders

The Kandy project targets three primary stakeholder groups, each with distinct needs that Kandy will address:

- **Administrators** who oversee Kafka performance, data integrity, and cluster health. Kandy's real-time monitoring features provide quick insights into cluster status, helping administrators maintain stability and react swiftly to any issues.
- **Developers** who integrate Kafka into their applications and require efficient access to Kafka for testing, debugging, and troubleshooting. Kandy streamlines access to critical metrics, enhancing their productivity and reducing setup time for Kafka-related tasks.
- **DevOps teams** responsible for Kafka cluster infrastructure, uptime, and stability. Kandy's intuitive interface and essential monitoring tools enable DevOps teams to oversee cluster health without the need for complex GUI-based solutions, making management tasks more efficient.

To represent the target audience, I have identified a key user persona: **Sarah**, the Head of Engineering at Valerann. Sarah oversees multiple data-driven projects that rely on real-time Kafka streams, frequently monitoring consumer lags, topic performance, and system stability to ensure smooth operation for her teams. With extensive experience in both development and Kafka management, Sarah's insights are invaluable for evaluating Kandy's effectiveness. Sarah's feedback will help me to be sure if Kandy meets the practical needs of it's users.

[1] **Kafka** is a distributed streaming platform designed for processing, storing, and transmitting data in real-time. It provides a reliable mechanism for data streaming, enabling applications and services to exchange information and respond to changes instantly. Kafka is built to handle enormous volumes of data, offering high performance and fault tolerance through its distributed architecture.

1.3 Kafka Architecture

Throughout whole project, I will use terminology specific to Kafka's architecture to describe its components and functionalities. So this subsection dives into the foundational concepts of Kafka. For better understanding the key ideas behind Kafka and its functionality, let's break it down into its core components and how they work together.

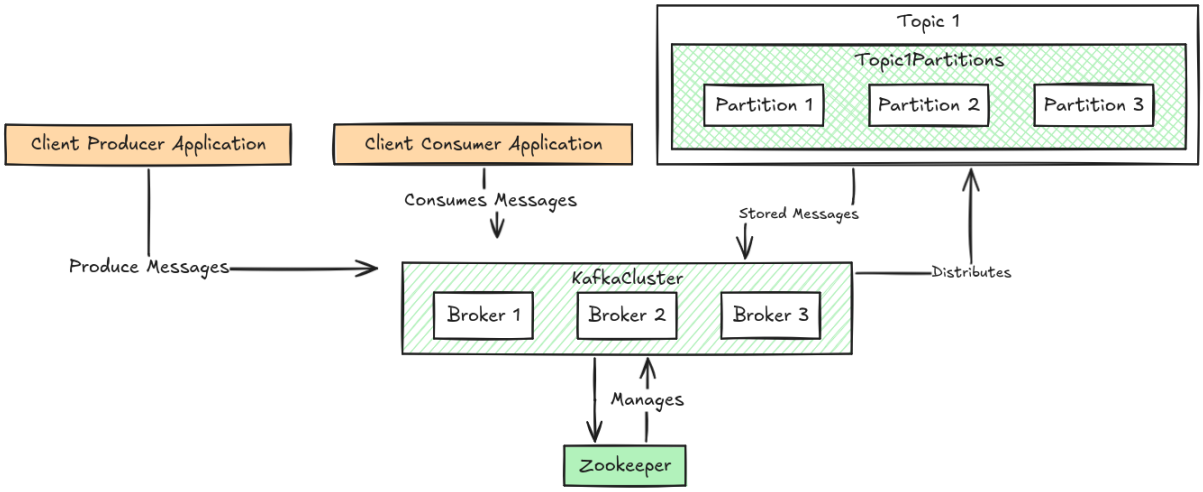


Figure 1.1: Key Components of Kafka Architecture

Core Components of Kafka

- **Message:** The fundamental unit of data in Kafka. It consists of a key, value, and optional metadata (headers)

  Example: A message in the *Weather_Updates* topic might include a key representing a city ID, a value containing temperature data, and headers indicating the time of measurement.

- **Producers:** These are the data generators. Producers are applications or systems responsible for sending messages to Kafka topics. Think of them as the "writers" in Kafka's ecosystem.

  Example: A weather monitoring application sending real-time temperature updates acts as a producer.

- **Consumers:** Consumers retrieve and process data stored in Kafka topics. They are the "readers" that transform raw data into actionable insights or visualizations.

  Example: A live dashboard showing weather updates would be a consumer.

- **Topics and Partitions:**

  - **Topics:** Logical containers that organize messages by category. For instance, a topic named *Weather_Updates* might store all weather-related data.
  - **Partitions:** Subsections of a topic that allow for parallel processing. Each partition stores messages in an immutable, sequential order. This setup enables Kafka to scale efficiently by distributing messages across multiple servers (brokers).

- **Brokers:** Brokers are Kafka's backbone. They store data and manage partitions. When combined, multiple brokers form a Kafka cluster, which offers high availability and fault tolerance by replicating and balancing data across servers.

- **Zookeeper:** Zookeeper is a distributed coordination service crucial to Kafka's operations. It tracks metadata, manages partition leadership, and monitors broker health to ensure cluster stability. Although Zookeeper is essential in Kafka's traditional setup, newer Kafka versions are moving towards its replacement with Kafka's internal *KRaft* system. *(Note: I'm not planning on working with Zookeeper directly in this project, but understanding its role might be helpful)*

How Kafka Works

1. **Producers Generate Data:** Applications acting as producers send data to Kafka, specifying the relevant topic. For instance, a weather app might publish updates to a *Weather_Updates* topic.
2. **Kafka Distributes Data Across Partitions:** Kafka divides incoming data into partitions within a topic and distributes these partitions across brokers in the cluster. This design allows efficient load balancing and supports simultaneous access by multiple consumers.
3. **Consumers Process Data:** Consumers subscribe to topics and retrieve data from specific partitions. Kafka ensures that data can be processed in parallel, enabling high-performance real-time analytics or other operations.

1.4 Computational Methods

Managing Apache Kafka, a distributed and data-intensive system, requires careful application of computational methods. Kafka's complex architecture involves intricate interactions between brokers, topics, partitions, and consumers. Computational approaches simplify this complexity, making it feasible to develop a Text-Based User Interface (TUI) that provides users with an intuitive way to manage Kafka. This section outlines how principles such as *Abstraction*, *Decomposition*, *Thinking Ahead*, and *Algorithmic Thinking* are employed in the development of the TUI.

1.4.1 Abstraction

Definition:

Abstraction simplifies a complex system by emphasizing its essential features while hiding less relevant details.

Application:

Abstraction is at the heart of this TUI's design, allowing users to interact with Kafka's core components—brokers, topics, partitions, and consumer groups—without requiring deep knowledge of Kafka's APIs or internal mechanisms. The TUI abstracts Kafka's functionalities into high-level commands and representations:

- **Brokers:** Represented as nodes with health status and connectivity information.
- **Topics:** Abstracted as logical containers for message streams, with partition and replication details available upon request.
- **Consumers:** Displayed as groups subscribing to topics, with key metrics such as consumer lag simplified for clarity.

This abstraction ensures that even novice users can perform critical Kafka operations, such as topic creation, monitoring, and deletion, using an accessible interface.

1.4.2 Decomposition

Definition:

Decomposition breaks a large, complex problem into smaller, manageable sub-problems.

Application:

The development of the TUI leverages decomposition to divide Kafka management into specific functional components:

- **Topic Management:** Includes operations such as creating, deleting, and listing topics with detailed configurations.
- **Consumer Monitoring:** Tracks consumer groups, displays subscription details, and monitors consumer lag.
- **Broker Status Monitoring:** Reports on broker health and displays connectivity details.

Each component is further divided into smaller tasks. For example, “Topic Management” includes sub-tasks like validating topic names, setting replication factors, and handling partition assignments. Decomposition makes the project more manageable, enables iterative development, and allows for parallelization of work across different modules.

1.4.3 Thinking Ahead

Definition:

Thinking Ahead involves anticipating future needs and ensuring the solution is robust and scalable.

Application:

The TUI is designed with extensibility in mind, anticipating potential changes in Kafka’s architecture and user requirements. For instance:

- The TUI supports dynamic configuration, allowing it to adapt to changes in the number of brokers or topics without significant modifications.
- Modular code design ensures that new features, such as support for Kafka Streams or advanced monitoring tools, can be integrated seamlessly.

By thinking ahead, the TUI ensures long-term usability and minimizes maintenance efforts, even as Kafka evolves.

1.4.4 Algorithmic Thinking

Definition:

Algorithmic Thinking involves designing and implementing efficient, logical steps to solve a problem.

Application:

Several features of the TUI rely on algorithmic thinking:

- Efficient algorithms for calculating consumer lag ensure that real-time metrics are displayed without significant delays.
- Partition assignment algorithms optimize how messages are distributed across brokers, ensuring load balancing.

- Sorting and filtering operations in the TUI enable users to quickly locate relevant topics or consumers, even in large Kafka clusters.

These algorithmic enhancements ensure that the TUI provides accurate and responsive feedback, critical for managing a live distributed system.

1.4.5 Evaluation and Refinement

As part of the computational process, evaluation and refinement ensure that the TUI meets user needs effectively. Continuous testing is performed to verify:

- Usability: Ensuring the interface remains intuitive for users with varying levels of experience.
- Performance: Verifying that the TUI scales efficiently with large Kafka clusters.
- Correctness: Ensuring that operations, such as topic creation or consumer monitoring, produce accurate results.

1.4.6 Conclusion

By applying computational methods such as abstraction, decomposition, thinking ahead, and algorithmic thinking, the TUI simplifies Kafka management for users. These principles ensure that the solution is user-friendly, scalable, and adaptable to future requirements, demonstrating the importance of computational methods in solving complex, real-world problems.

2 Research

2.1 Interview questions

I will outline here key questions that I will ask stakeholder, but during the interview I may come up with follow up questions. The interview will help me find their opinion on the software and how they would like to use it.

2.1.1 Questions

- **How often do you work with Kafka? What utilities do you use to work with it? Why those?**
  - Understanding the frequency of interaction with Kafka is crucial for designing a utility that meets the stakeholder's needs without overwhelming them with unnecessary features
- **Can you describe in detail the last problem you solved related to Kafka?**
  - Gathering detailed information about recent challenges helps in designing features and functionalities within the utility to address common pain points
- **Is there anything you're currently dissatisfied with in the utilities you use? What would you change?**
  - Discovering dissatisfaction with existing utilities guides the development of the utility by focusing on areas for improvement or enhancement
  - Knowing desired changes informs feature prioritisation, ensuring that the utility aligns with stakeholder expectations and addresses their specific pain points effectively
- **Do you have a preference for a Terminal User Interface (TUI) or web-based solution for managing Kafka? If so, what are the reasons behind your preference?**
  - Understanding stakeholders' preferences for either a Terminal User Interface or web-based solution provides crucial insights into their workflow preferences and the environment in which they operate. This information ensures that the utility is developed in a manner that seamlessly integrates into their existing workflows, ultimately enhancing user experience and satisfaction
- **Describe the recent problems you encountered while using ¡Name of the utilities¿**
  - Identifying areas of dissatisfaction with current utilities allows for targeted improvements in the design and functionality of the TUI utility. By addressing these pain points, the TUI utility can be tailored to better meet the needs and expectations of stakeholders, resulting in a more effective and user-friendly solution.
- **What do you believe would be the ideal number of clusters the system should handle simultaneously? Why? How many clusters have you worked with simultaneously in the past?**
  - Understanding the optimal number of clusters for concurrent support is key to ensuring the system scales seamlessly and manages resources efficiently.
- **Could you explain how you currently configure connections to the clusters, and how user-friendly do you find this process?**

  - Understanding how connections to the clusters are currently configured and evaluating the user-friendliness of this process can provide valuable insights, ensuring that the new utility addresses any shortcomings and provides a seamless user experience.

- **When thinking about how data from the clusters is displayed, what are your main expectations? For example, how quickly data is presented by current tools, and how satisfied are you with this?**
  - Understanding the expectations regarding data display from the clusters is crucial for informing development priorities and ensuring alignment with stakeholder needs and by gaining insights into these aspects, we can effectively prioritise features and enhancements within the utility

- **Can you describe some common scenarios where you would use multiple clusters? For instance, comparing data across different topics or any other important scenarios you have in mind?**
  - Understanding common scenarios where users would use multiple clusters is important for designing a utility that meets users' needs effectively. By considering these scenarios, I can ensure that the utility is versatile enough to support a wide range of users workflow needs

2.2 Interview

Can you please introduce yourself and describe your experience with Kafka?

Sure. My name is Sarah Anderson, and I'm the Head of Engineering at Valerann. I've been with the company for nearly three years, and I've been working in tech for almost ten years. Regarding my experience with Kafka, I started using it about six years ago while working at a marketing technology company. They used Kafka to process large volumes of email notifications and automatically convert them into sales. Since then, I've worked with Kafka in about five different companies, all primarily for high-volume, real-time data processing.

Typically, my role with Kafka has involved helping companies implement and scale it from the ground up. Kafka provides excellent scalability, but transitioning from a traditional monolithic architecture to an event-driven, Kafka-based design requires significant effort. Developers often need training, and management needs to understand how to structure and scale a Kafka architecture. I've spent much of my time helping teams learn how to use Kafka, build systems around it, and implement it effectively.

How often do you work with Kafka now, and what utilities and tools do you use to manage it?

Currently, I work with Kafka every few days—around three times a week. However, I don't interact with the individual services as much as I used to. In terms of tools, we primarily use a tool called Faust Streaming, which was derived from an older tool called Faust (created by a company called Robinhood, not to be confused with the Bitcoin company). However, Faust is now deprecated, and we're moving away from it. We've been using it as our streaming framework, similar to the Kafka Streams Java framework but written in Python—and not as efficiently. We're gradually replacing it with an in-house tool we've developed called CitizenK (inspired by the movie *Citizen Kane* and Kafka's initial).

For managing Kafka day-to-day, we use an open-source tool from Provectus Labs that provides a free UI for administering Kafka clusters, performing tasks like cleaning up consumer groups or resetting offsets. For infrastructure management, we use Terraform to create clusters, brokers, topics, and set up initial settings.

Can you describe any problems you've encountered while using these tools to manage Kafka?

One of the most common issues we face with admin tools is the lack of effective consumer group management. No tool I've found manages consumer groups as well as the Kafka CLI, but the CLI is cumbersome to use on a daily basis since it requires running bash scripts. For example, if you want to remove a topic from a consumer group, no admin tool I know of can do that; they only allow you to delete the entire consumer group or reset offsets. The CLI supports this, but no admin tool does, which is frustrating.

Do you prefer web tools over CLI tools? Why?

I prefer web tools because they make onboarding new users much easier. It's harder to get people to read through extensive documentation to learn CLI tools. With a web interface, you can include visual guides like screenshots with instructions such as "click this button," which is much more intuitive. Writing documentation for CLI usage, such as which arguments to use and when, is far more complex. Admin UIs also offer the ability to limit user permissions, which is helpful. For example, tools like Conductor allow you to create user accounts with different levels of permissions, whereas with the CLI, users can do almost anything once they have access.

Would you prefer a Terminal User Interface, like Lazydocker, or a Web Interface?

Personally, I prefer a web interface, although I know many developers prefer terminals. I'm one of the rare developers who enjoys working with a web interface. However, a terminal interface could benefit many people, especially if it's easier to use than the Kafka CLI. A simplified terminal interface that eliminates the need to reference various configuration files and allows everything to be executed from a single root command would be a vast improvement.

Can you describe your experience working with multiple Kafka clusters simultaneously?

We currently have seven Kafka clusters, each with both staging and production environments, which are differentiated only by name. Managing these clusters can be challenging. It would be great to have a tool that allows you to scan messages across all clusters or filter topics across clusters. Managing consumer groups across multiple clusters is also a manual process. For instance, resetting the offset of a consumer group in every cluster requires going through each cluster one by one, which is time-consuming and error-prone.

How do you connect to Kafka clusters, and how user-friendly is the process?

We connect to Kafka clusters in three main ways. First, for all our services running in AWS, we use an AWS secret that contains Kafka credentials. These credentials are global, shared across all services for a particular customer. This makes things easier, although we recognize that creating separate credentials per service is something we should do but haven't yet automated.

Second, we use an admin UI, which runs on local machines rather than in the cloud. We have an internal CLI tool that starts the Kafka UI, reconnecting it to all our clusters so we can browse them through the interface.

Third, we interact with Kafka through the CLI. We have another internal CLI tool that generates Kafka configuration files using the AWS global secret.

What are your expectations for how data from clusters, such as topics and consumer groups, should be displayed?

I think flexibility is key. It would be great to customize the way data is displayed, for example, by pulling out information for a specific topic across all clusters or viewing all topics for a particular cluster. Additionally, showing statistics like the minimum and maximum lag for a particular topic across clusters, or the average size of topics, would be very useful.

Are you satisfied with the current tools' representation of this data?

No, I'm not satisfied at all. Most tools are awkward to use. They typically have separate subsections for topics and consumer groups, but navigating between them is cumbersome, and you often lose track of where you are in the interface.

Can you describe the last Kafka-related problem you solved?

The last Kafka-related problem I solved was deleting topics and consumer groups. Since we use Amazon MSK to host Kafka, AWS handles most of the Kafka-specific issues that arise from maintaining clusters. Our biggest challenges are usually administrative, like keeping Kafka topics up-to-date as we change their structure and settings.

How important is it to sort topics by different parameters in your daily tasks?

It would be extremely useful to sort topics by parameters like message volume or data size. Many of our services share topics, so sometimes, a service will start producing a large number of messages to a shared topic. Being able to quickly identify which topics have the highest message count or data volume would help us stay on top of this. Unfortunately, our current admin UI only allows us to sort topics by name, which isn't very helpful.

What are the top three features you'd like to see in a Kafka admin UI tool?

- Better management of consumer groups that supports all the features available in the Kafka CLI.
- Cross-cluster topic statistics, so we can compare topics across multiple clusters and see metrics like size, number of connections, etc.
- The ability to limit user permissions. For instance, I don't want everyone to be able to delete topics; only a few people should have that capability. It would be helpful if the UI could enforce this, in addition to Kafka's own security settings.

Insights from Interviews: Key needs identified include comprehensive consumer group management, cross-cluster statistics, and user permissions. These priorities will directly inform Kandy's development, focusing on user-centered functionality improvements.

2.3 Research on Existing Solutions

To develop an effective Kafka management tool that aligns with the needs of Kafka administrators and engineers, I conducted a research of existing solutions across GitHub, Habr, Reddit threads focused on Kafka, and various package repositories. This research aimed to identify current tools' strengths and limitations, assess their suitability for a secure, scalable environment.

2.3.1 Criteria for Evaluating Usability

To assess each tool's effectiveness, I developed the following criteria, informed by Sarah's preferences and project objectives

1. **Open Source**: Determines the ability to customize and scale the tool affordably in an enterprise environment.
2. **Cost**: Evaluates cost-effectiveness for deployment at scale, balancing proprietary options against open-source flexibility.
3. **Interface Type**: The tool's user interface—CLI, TUI, or web-based—affects usability, with a preference for web-based tools for accessibility and TUI for optional flexibility.
4. **Consumer Group Management**: Ability to manage consumer groups, including tasks like targeted offset resets, affects the tool's effectiveness in data-driven operations.
5. **Cross-Cluster Capabilities**: Supports management across multiple Kafka clusters, vital for Sarah's team managing complex deployments.
6. **Data Sorting and Flexibility**: Ability to sort topics by message volume, data size, and other metrics for an optimized user experience.
7. **User Permissions**: Provides secure, granular user roles for safe, role-based access management in multi-user environments.

2.3.2 Kafka Manager

Overview

Kafka Manager, a community-maintained open-source tool by Yahoo, offers a basic web-based interface for managing Kafka clusters. It provides standard functionality for topic visibility and consumer group management.

Features and Limitations Based on Criteria

- **Open Source**: Yes
- **Cost**: Free
- **Interface Type**: Web
- **Consumer Group Management**: Limited; lacks granular control such as the ability to remove specific topics from a consumer group, an important feature for custom data pipelines.
- **Cross-Cluster Capabilities**: Lacks multi-cluster management; suitable only for single-cluster setups, making it less viable for larger, distributed teams.
- **Data Sorting and Flexibility**: Limited sorting options; topics are sortable by name only, which restricts dynamic data organization.
- **User Permissions**: Lacks user-role management, posing a security concern in multi-user environments.

[Placeholder for image: Screenshot of Kafka Manager's interface displaying a single cluster's topic and consumer group information.]

Suitability Summary: Kafka Manager's lack of cross-cluster capabilities and limited consumer group control reduce its practicality for Sarah's team. Its single-cluster design makes it less scalable, while the absence of user permissions limits security.

2.3.3 Conduktor

Overview

Conduktor, a proprietary Kafka management tool, offers a robust web-based GUI with advanced querying capabilities. Its features cater to a broad range of Kafka ecosystem components, providing operators with enhanced flexibility.

Features and Limitations Based on Criteria

- **Open Source**: No
- **Cost**: Proprietary; requires licensing, which may reduce cost-effectiveness at scale.
- **Interface Type**: Web
- **Consumer Group Management**: Provides good consumer group controls but lacks the granularity found in CLI tools for highly specific data manipulation.
- **Cross-Cluster Capabilities**: Supports multiple clusters but lacks cross-cluster statistics, which limits data insights across clusters.
- **Data Sorting and Flexibility**: Offers flexible sorting by various metrics; however, it does not support cross-cluster topic comparisons, limiting use in environments needing aggregated insights.
- **User Permissions**: Includes role-based access controls, a core requirement for Sarah's team.

Suitability Summary: Conduktor's robust interface and user-permission management are valuable; however, its proprietary model and limited cross-cluster statistics make it less adaptable for scalable, cost-effective Kafka deployments.

[Placeholder for image: Conduktor's GUI showing user permissions settings and multi-cluster management options.]

2.3.4 Kafdrop

Overview

Kafdrop is an open-source, web-based tool for monitoring Kafka clusters. It provides a straightforward UI, focusing on Kafka brokers, topics, and partitions, making it suitable for small to medium clusters.

Features and Limitations Based on Criteria

- **Open Source**: Yes
- **Cost**: Free
- **Interface Type**: Web
- **Consumer Group Management**: Basic; lacks advanced control such as offset resets for specific consumer groups.
- **Cross-Cluster Capabilities**: Not supported; limited to monitoring single clusters.
- **Data Sorting and Flexibility**: Limited sorting options; cross-cluster comparisons absent.
- **User Permissions**: Does not support role-based access control, which reduces its security in multi-user scenarios.

[Placeholder for image: Kafdrop's UI displaying basic cluster status and topic information.]

Suitability Summary: While user-friendly, Kafdrop's limited features make it unsuitable for large-scale Kafka operations, especially those requiring in-depth consumer group management and cross-cluster capabilities.

2.3.5 Kafka CLI Tools

Overview

Kafka CLI Tools, built directly into Apache Kafka, provide a set of powerful, low-level commands for various management tasks, including managing topics and consumer groups.

Features and Limitations Based on Criteria

- **Open Source**: Yes
- **Cost**: Free
- **Interface Type**: CLI
- **Consumer Group Management**: Comprehensive; provides granular control over offsets, including removing specific topics from consumer groups.
- **Cross-Cluster Capabilities**: Limited; each cluster requires manual configuration.
- **Data Sorting and Flexibility**: Limited; lacks built-in sorting and visualization without custom scripting.
- **User Permissions**: No user-role management, complicating restricted access in shared environments.

Suitability Summary: Kafka CLI Tools provide high control but lack usability for daily operations, visualizations, and multi-user management, making them unsuitable for teams requiring intuitive interfaces and cross-cluster management.

[Placeholder for image: Example of Kafka CLI commands being executed to manage topics and consumer groups.]

2.3.6 kafkactl

Overview

kafkactl, an open-source CLI tool inspired by Kubernetes' `kubectl` syntax, is designed for teams familiar with Kubernetes looking to streamline Kafka management.

Features and Limitations Based on Criteria

- **Open Source**: Yes
- **Cost**: Free
- **Interface Type**: CLI
- **Consumer Group Management**: Moderate; lacks full Kafka CLI capabilities.
- **Cross-Cluster Capabilities**: Not inherently supported; can be configured.
- **Data Sorting and Flexibility**: Limited; requires additional scripting for data sorting.
- **User Permissions**: No role-based management, limiting security in multi-user settings.

[Placeholder for image: Terminal interface showing kafkactl commands managing Kafka topics in a Kubernetes-style syntax.]

Suitability Summary: kafkactl is more suited to teams with existing CLI expertise but lacks essential user permissions and cross-cluster capabilities for complex deployments.

2.3.7 Kafka Monitor

Overview

Kafka Monitor, an open-source tool by LinkedIn, provides monitoring for Kafka performance, primarily focusing on end-to-end latency, throughput, and reliability.

Features and Limitations Based on Criteria

- **Open Source**: Yes
- **Cost**: Free
- **Interface Type**: CLI-based monitoring
- **Consumer Group Management**: Limited; primarily a monitoring tool.
- **Cross-Cluster Capabilities**: Lacks cross-cluster monitoring.
- **Data Sorting and Flexibility**: Primarily focused on performance metrics with minimal interactive sorting.
- **User Permissions**: No role management, limiting secure multi-user access.

[Placeholder for image: Graph output from Kafka Monitor displaying latency and throughput metrics.]

Suitability Summary: While beneficial for performance monitoring, Kafka Monitor lacks management and cross-cluster features needed for daily Kafka administration.

2.3.8 Comparative Summary

Tool	Open Source	Cost	Interface	Consumer Group	Cross-Cluster	Data Sorting	Permissions
Kafka Manager	Yes	Free	Web	Limited	No	Minimal	No
Conduktor	No	Proprietary	Web	Good	Limited	Flexible	Yes
Kafdrop	Yes	Free	Web	Basic	No	Minimal	No
Kafka CLI Tools	Yes	Free	CLI	Comprehensive	Manual	Minimal	No
kafkactl	Yes	Free	CLI	Moderate	Configurable	Limited	No
Kafka Monitor	Yes	Free	CLI	Limited	No	Minimal	No

Table 2.1: Comparison of Kafka Management Tools by Feature

After gathering this information, it's evident that Conduktor largely addresses Sarah's requirements.[1] It would be ideal to explore its functionality further; however, due to its proprietary nature, I was unable to do so.

The next best options to consider for development are Kafka CLI Tools and Kafka Manager. Kafka CLI Tools were developed by Confluent, the creators of Kafka, and offer a full suite of utilities to manage Kafka-related issues. However, based on Sarah's feedback and my own experience, Kafka CLI Tools can be challenging to use due to the numerous commands and flags, which are not always intuitive. Kafka Manager, on the other hand, is a free alternative to Conduktor, supporting all essential features needed for cluster management.

[1] As I later learned, **Conduktor** is indeed the solution Sarah's team uses in their development process.

2.4 Look into Other TUI Applications

My experience in UI development is close to zero, so I decided to draw inspiration and knowledge from TUI wrappers for other services, such as:

lazydocker: A TUI for managing Docker containers

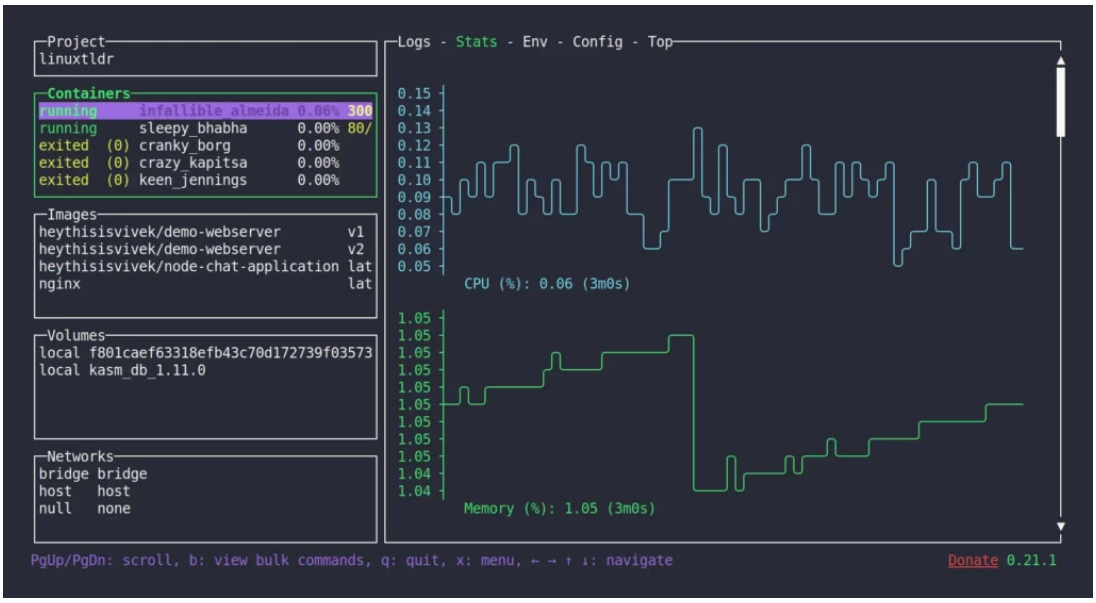


Figure 2.1: LazyDocker main screen

Dolphie: A utility for SQL database analytics

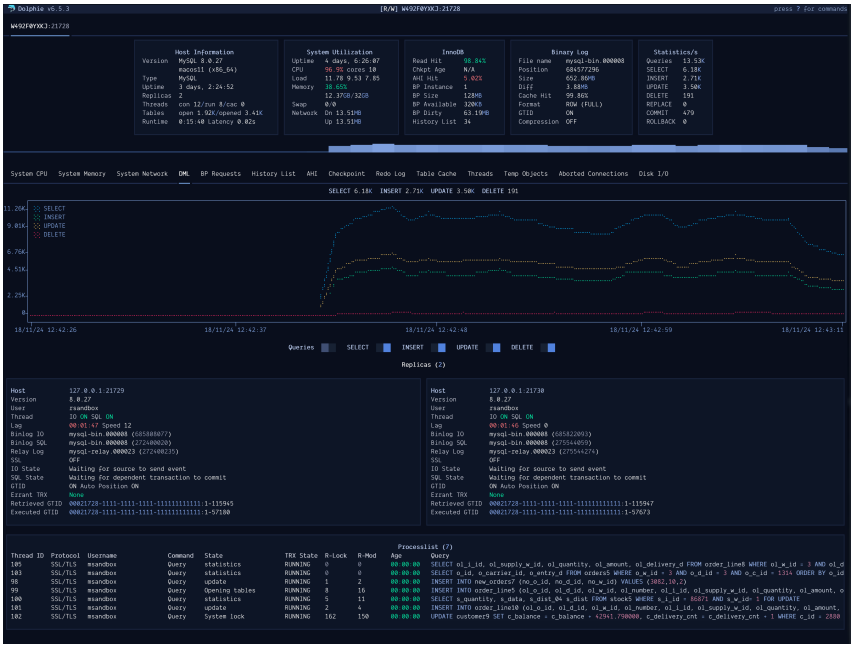


Figure 2.2: Dolphie main screen

These are tools I personally use daily and find convenient. They allow me to quickly access necessary information.

2.4.1 LazyDocker Analysis

To understand how exactly these tooks work, I analyzed LazyDocker.

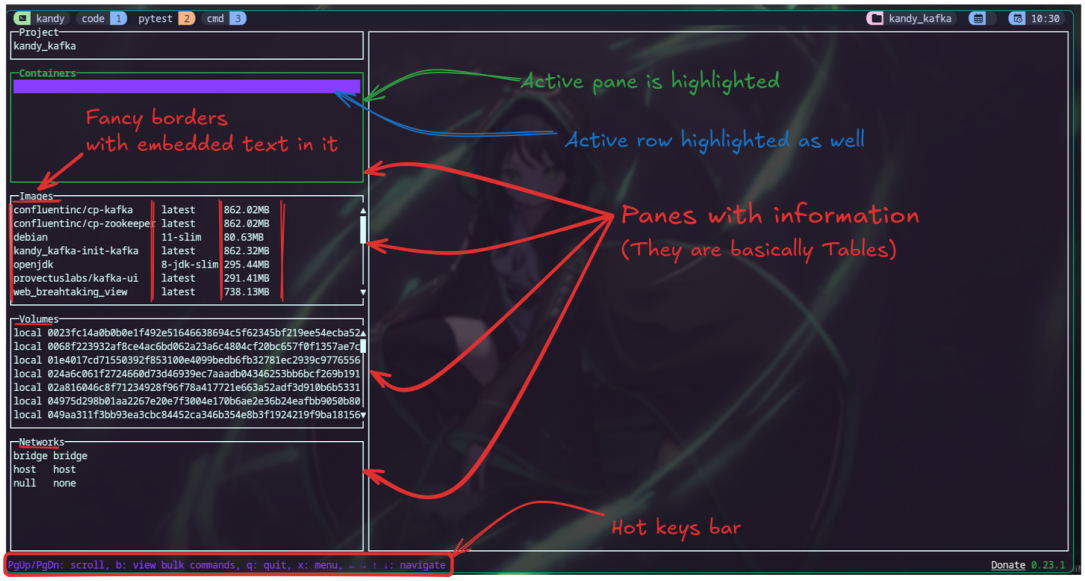


Figure 2.3: Lazydocker main screen analysis

When launching the utility, we can see the compact and minimalist design. All necessary categories are divided into panes, and navigation between them is performed by pressing `Tab`/`Shift+Tab` or using arrow keys. At the bottom of the screen, there's also a bar with shortcut keys.

Panes are the most critical component of this interface. They display information, and their layout heavily influences the user experience. In the case of LazyDocker, the panes work as tables or lists of options. Beyond names, they show essential information, allowing users to navigate without guessing and compare container performance without opening detailed logs.

Additionally, there's one large pane with detailed information about the selected container/image or network. This pane occupies more than half the screen, providing ample space for detailed data.

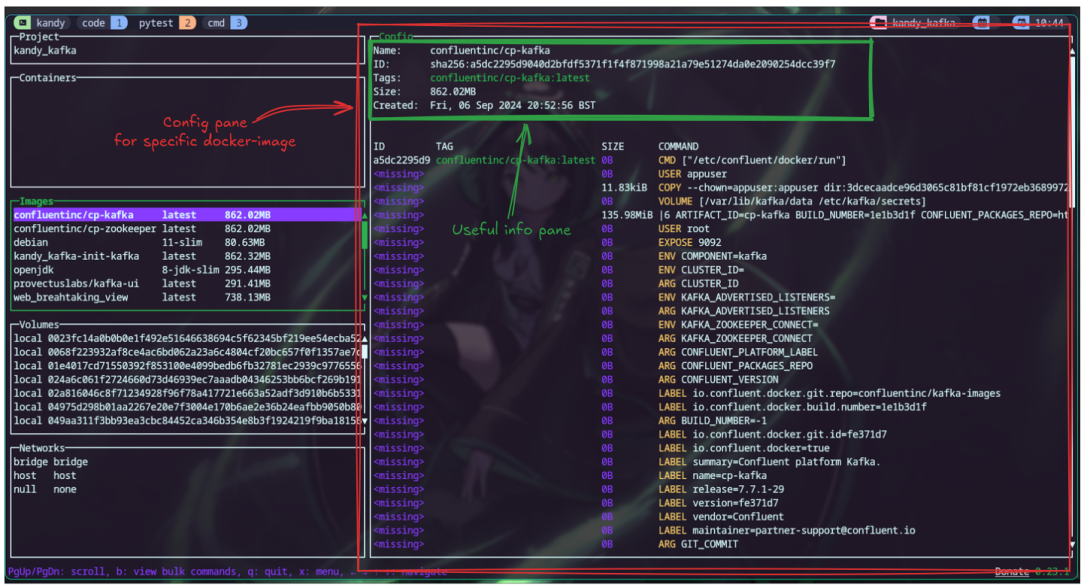
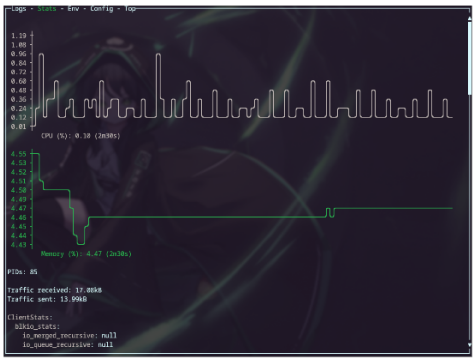


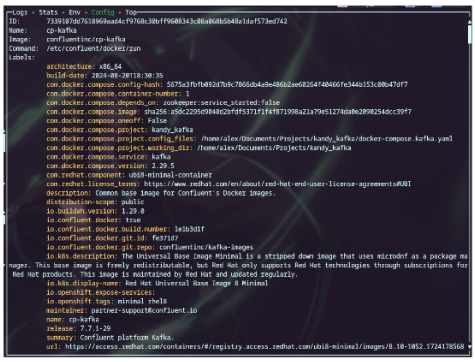
Figure 2.4: LazyDocker: Config pane

This pane is divided into several tabs. Besides configuration tab (2.4), you can view statistics for each container (2.5a), environment variables (2.5b), and resource usage (2.5d). This pane is the primary and most convenient feature for day-to-day Docker management. I believe a similar approach could be applied to my product.



(a) Stats

(b) Env



(c) Config

(d) Top

Figure 2.5: Big pane tabs

Lastly, pressing a hotkey `x` opens a menu with a wide range of commands. Hotkeys are designed to speed up processes, and this kind of menu, which consolidates all necessary commands, significantly enhances efficiency. Unlike the bar, this menu allows for a detailed description of each key binding.

This is probably my favorite feature of LazyDocker because it eliminates the need to exit the utility and type lengthy commands to accomplish simple tasks and basiclly speeds up my work several times. I definitely plan to implement such a menu in my product.

2.4.2 Conclusions

After analyzing LazyDocker, I identified key features that could be useful in creating my own utility:

- **Pane-based layout**: This structure looks harmonious and logical for such utilities. Each pane functions as a container, simplifying the logic of information display.
- **Table-like panes**: Displaying data in this format aligns with Kafka's structure and is likely the optimal solution.
- **Menu and hotkey bar**: These elements make navigation through the interface a quick and effortless task, often requiring no conscious thought.
- **Tabbed panes**: As with the large pane, using tabs instead of separate screens can sometimes be more convenient and logical.

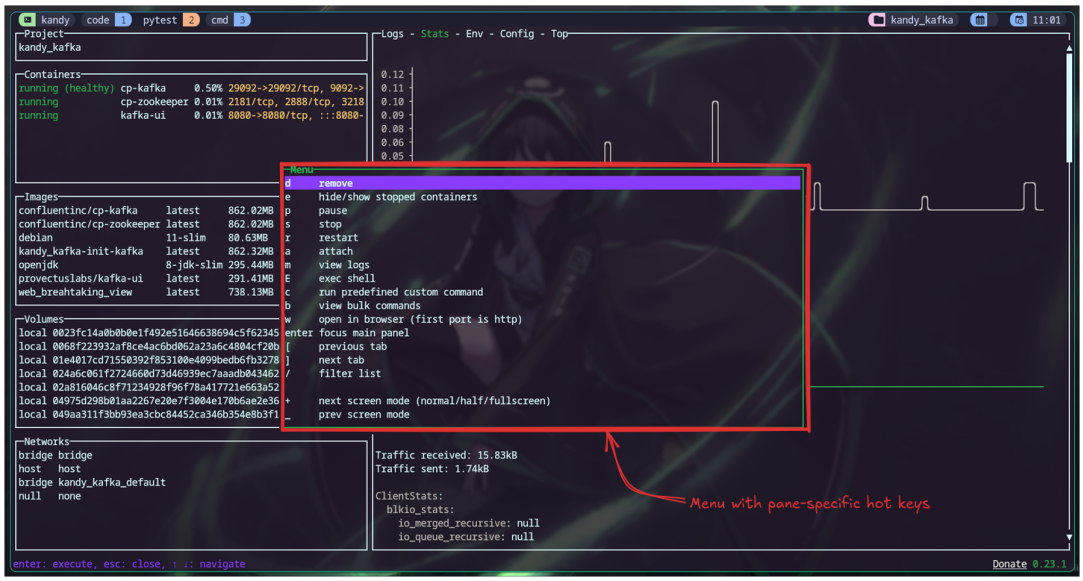
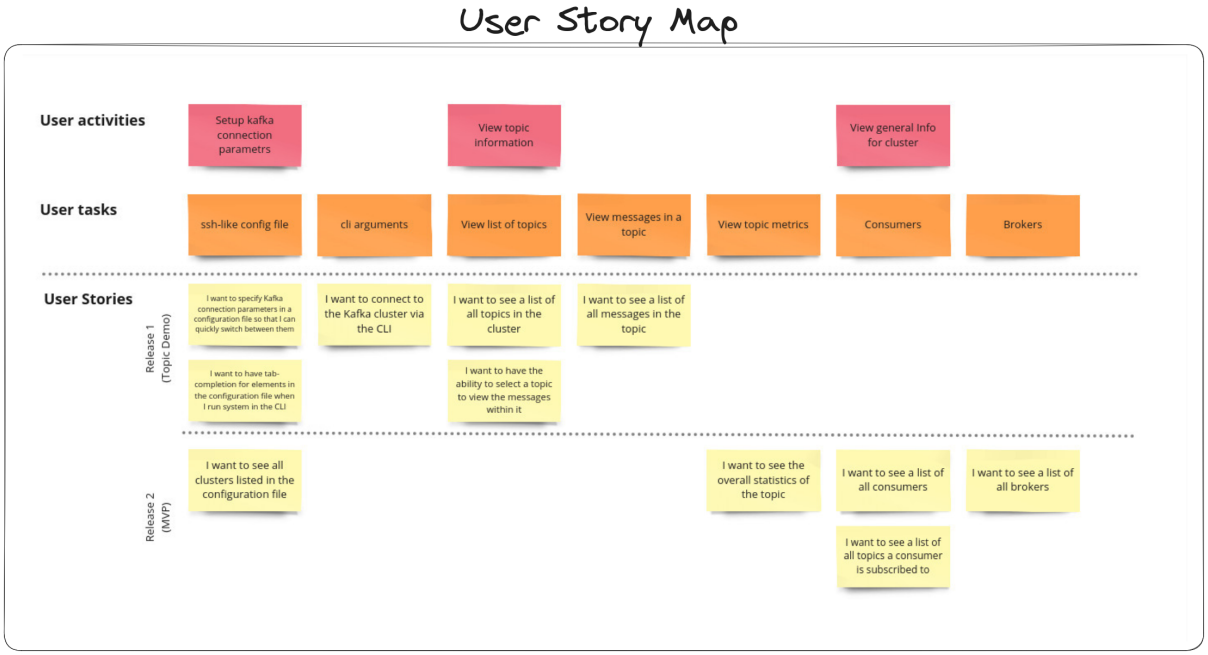


Figure 2.6: Hotkey Menu

- **Colors**: Though not explicitly mentioned earlier, colors play a vital role both aesthetically and functionally. Highlighting the active pane, using a “traffic light” system to display container statuses, and syntax highlighting for configurations all enhance the user experience.

By incorporating these features, I aim to develop a TUI application that is both efficient and user-friendly.

2.5 Features of the proposed solution



To outline the initial application concept and plan for future development, I chose to use User Story Mapping (USM). This approach provides a user-centered perspective, aiding in prioritizing features across different development phases. USM is an agile approach to requirements engineering, composed of several key elements:

1. **User Activity:** This step involves identifying the main activities users will perform within the system, representing the core actions they take to achieve their goals.
2. **User Tasks:** Within each user activity, specific tasks are defined. These tasks represent the necessary steps users must follow to accomplish the activity.
3. **User Stories:** Each task is further refined into user stories. These stories represent the functionalities or features needed to fulfill the task from the user's perspective.

In a User Story Map, user stories are organized into swimlanes based on their release phases. Releases indicate the prioritization of user stories, outlining which ones should be completed first. The essential structure of USM lies in its sequential arrangement of user tasks from left to right, aiming to provide a comprehensive overview of user interactions with the system. This format helps ensure a clear understanding of user needs and workflows.

One of the main advantages of USM over other requirements engineering techniques is its flexibility to accommodate changes based on user feedback. Its visual representation also aids in understanding the system from the user's perspective, enhancing communication and collaboration among stakeholders.

On the map, tasks are divided into two releases: Topic Demo and MVP. The Topic Demo represents the initial stage, where I will create the application framework, TUI, and add the essential topic-viewing feature. This demo serves as a basis for consulting with my stakeholder and allows for adjustments to the USM if needed.

After consulting with Sarah, I will refine the application according to her feedback and proceed to work on the second release, MVP (Minimal Viable Product). This release will include additional features, such as displaying statistics across all topics and enabling interaction with consumer groups.

2.6 Limitations of my solution:

The primary limitation of this solution is time. As this project is part of the A-level coursework, alongside other assignments, my task estimates may be overly optimistic. Working independently, I may realistically only be able to implement a portion of the planned functionality.

Additionally, there are limitations associated with the libraries I intend to use. Currently, I am considering urwid or textual for developing the TUI (Text User Interface). However, urwid, for instance, does not support a broad range of terminal types and may perform inconsistently in certain environments, which could pose challenges and restrict my solution's effectiveness.

3 Requirements

3.1 Software product quality model

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS FUNCTIONAL CORRECTNESS FUNCTIONAL APPROPRIATENESS iso25000.com	TIME BEHAVIOUR RESOURCE UTILIZATION CAPACITY	CO-EXISTENCE INTEROPERABILITY	APPROPRIATENESS RECOGNIZABILITY LEARNABILITY OPERABILITY USER ERROR PROTECTION USER ENGAGEMENT INCLUSIVITY USER ASSISTANCE SELF-DESCRIPTIVENESS	FAULTLESSNESS AVAILABILITY FAULT TOLERANCE RECOVERABILITY	CONFIDENTIALITY INTEGRITY NON-REPUDIATION ACCOUNTABILITY AUTHENTICITY RESISTANCE	MODULARITY REUSABILITY ANALYSABILITY MODIFIABILITY TESTABILITY	ADAPTABILITY SCALABILITY INSTALLABILITY REPLACEABILITY	OPERATIONAL CONSTRAINT RISK IDENTIFICATION FAIL SAFE HAZARD WARNING SAFE INTEGRATION

I based my quality requirements on the ISO 25010 standard, which outlines the key characteristics of any software product, and developed a quality model using the QAS (Quality Attribute Scenarios) methodology. To achieve this, I identified critical attributes by referencing both interviews with Sarah and industry standards. A description of the model and its attributes is provided below.

3.1.1 Quality Attribute Scenarios (QAS)

Each quality attribute is defined through quality attribute scen-arios, which must adhere to the six-part rule. For more information on this method, see https://wstomv.win.tue.nl/edu/2ii45/year-0910/Software_Architecture_in_Pr

Term	Description
Stimulus	Describes an event arriving at the system, like a performance event, user operation, or security attack. A stimulus for modifiability could be a modification request, while for testability, it could be a phase completion.
Stimulus Source	The origin of a stimulus, affecting how the system treats it. For example, a trusted user's request may be handled differently from an untrusted user's.
Response	Specifies how the system or developers should act in response to the stimulus, detailing responsibilities at runtime or during development.
Response Measure	Defines how to judge if the response meets requirements, helping determine if the system or developer actions are sufficient.
Environment	The context in which the scenario occurs, which qualifies the stimulus and influences system response.
Artefact	The part of the system to which the requirement applies, often the entire system but sometimes only specific components.

Table 3.1: Definitions of Key Terms

3.1.2 Environment

I am gonna use my laptop as a **normal operating conditions** for several reasons. First of all, it's always at hand, so I don't need to worry about logistics or where I'll be testing the product. Secondly, I have a fairly average laptop in terms of specifications, so if the utility works smoothly on it, it will likely work on more powerful machines as well. Therefore, all hardware requirements are based on the basic characteristics of my device and all software requirements are based on the installed software on the my machine.

Part of environment	Variable
OS	Manjaro Linux 23.1.4 (Vulcan)
Terminal Emulator	Kitty 0.31.0
Kafka version	confluentinc/cp-kafka:7.7.1
Python version	3.11.4
CPU	1.60 GHz
RAM	4 GB
Docker version	25.0.3
Python libs versions	Specified in the file **pyproject.toml** in Appendix:B.1

As Kafka will be deployed on the same system where Kandy is tested, I anticipate virtually negligible latency when it comes to communicating between services

3.2 Success criteria

Criteria	Justification	Evidence
Time behaviour QAS have passed successfully (Response time less than 10 seconds https://www.nngroup.com/articles/response-times-3-important-limits/)		.log file

4 Design

Having identified the requirements and success criteria, I can now proceed with designing the system itself. The primary goal of this process is to create a structure that facilitates the addition of new features and the maintenance of the project while preserving or improving its functionality. A thoughtful design approach is essential to ensure effective system development, maintainability, scalability, and adaptability for future changes.

Any design process begins with the decomposition of the problem into manageable pieces of work and domain-specific elements. This activity often involves brainstorming.

As a visual thinker, I decided to create a diagram that depicts the elements and sections of the system being designed.

4.1 Decomposition

The first level of elements I identified is as follows:

- Define core functional requirements
- Choose development technology
- Design user interface structure
- Test and debug
- Release and support

Since I am building a tool for an existing software system, my tool is heavily dependent on its core functionality and capabilities. Therefore, understanding and defining the core functional requirements is a high-priority activity.

Choosing suitable development technology is critical for several reasons. First, it must enable technical integration with Kafka. Second, I must have sufficient expertise in the chosen technology to ensure that the project is completed within the set timeline.

Given that the project goal is to create an application with a user interface (UI) to enhance how target users interact with Kafka, I need to carefully plan the UI's structure and design to ensure an optimal user experience.

Any software application requires an appropriate level of testing. To define what "appropriate" means for my application, I must consider how the system will be tested and determine the level of logging required to ensure that it works as expected.

The final phase of application development involves delivering the software to the target users. The process of installation and running the application is just as important as the application requirements themselves. Thus, I aim to plan ahead and identify the tools and strategies needed to make this process as seamless as possible.

In the following sections, I will explain my thought process for each element and how they contribute to the system's development.

4.1.1 Prioritization

Features are categorized into **essential** (green), **desirable** (yellow), and **optional** (red):

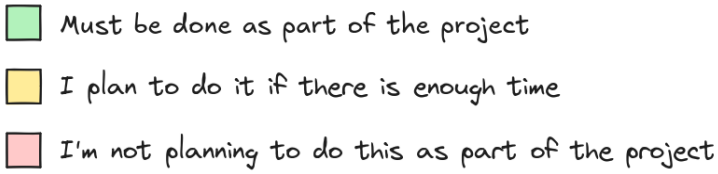


Figure 4.1: System Design Diagram Legend

For the full diagram, refer to **Appendix A**. Given the large size of the diagram, I will discuss its components and subdivisions separately in the following sections

4.1.2 Core functional requirements

As mentioned earlier, understanding and defining the core functional requirements is a high-priority activity. I have divided this into three main blocks:

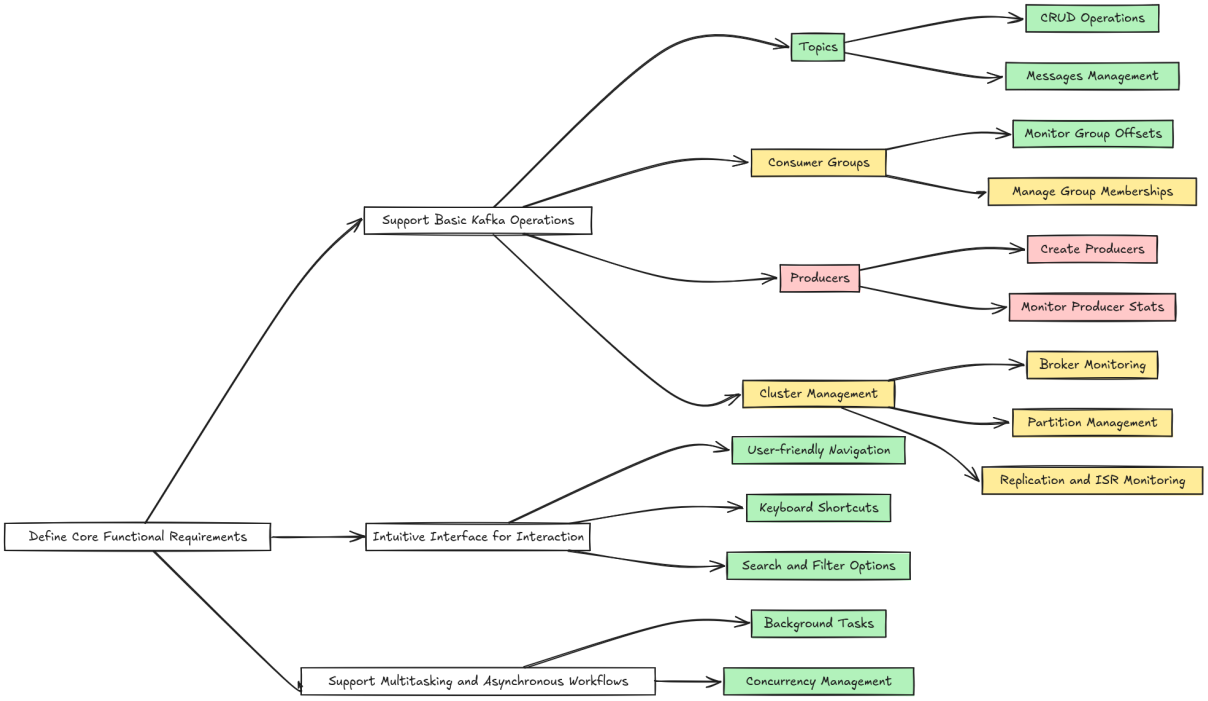


Figure 4.2: Core Functional Requirements

- **Basic Kafka Operations** — This involves core functionalities for interacting with Kafka. These are features that are essential for any Kafka management utility. I have outlined them separately in Figure 4.4, which I will discuss further below.
- **Intuitive Interface** — Since I am developing a terminal-based utility for developers who are accustomed to working with terminal applications like Vim, creating user-friendly navigation through versatile keyboard shortcuts is a priority. Additionally, because Kafka is inherently designed for processing large volumes of data, it is essential to provide users with the ability to quickly search for relevant information within a large data stream.

- **Multitasking and Asynchronous Workflows** — Again, considering Kafka's nature of handling significant amounts of data, the fetching process can take time. Ensuring concurrency of processes in this scenario is necessary, and implementing background tasks will be an effective solution.

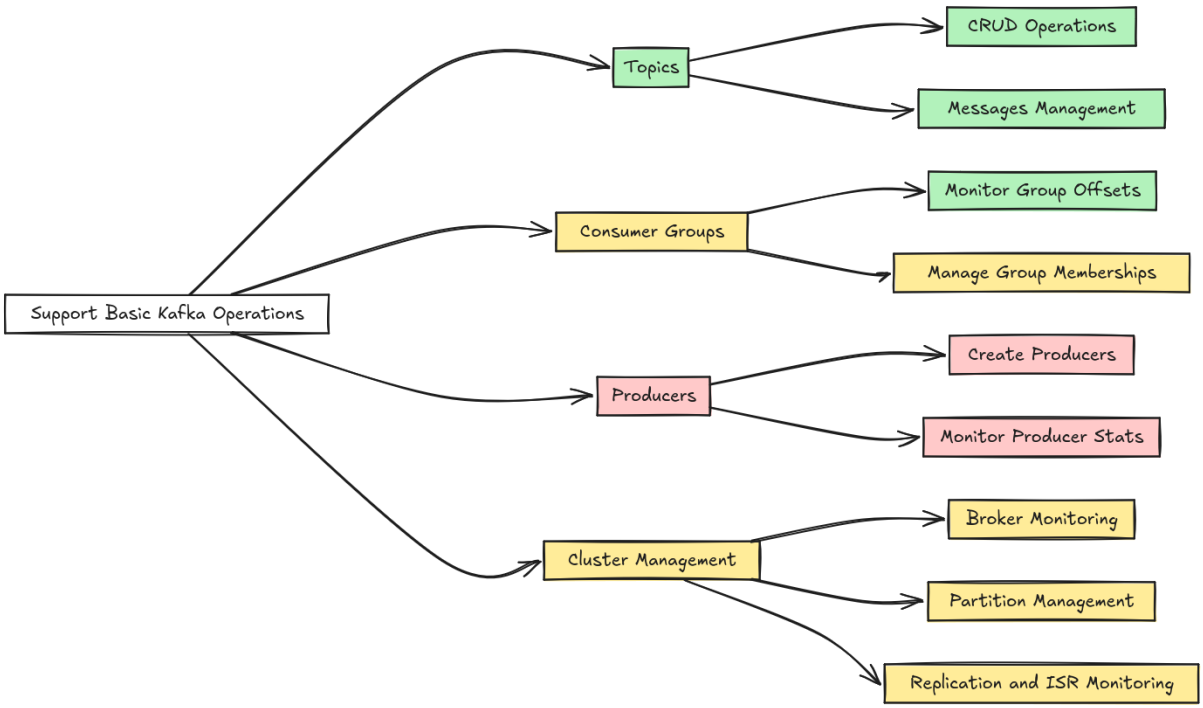


Figure 4.3: Basic Kafka operations requirements

Based on Kafka's architecture (Figure 1.1), four primary features come to mind that the management utility must support:

- Working with topics
- Managing consumer groups
- Interfacing with producers
- Cluster management

4.1.3 Interface Structure

The technologies chosen for UI development significantly influence the design process. Therefore, I want to finalize the interface design before selecting the technology stack. In Figure 4.5, I have highlighted the priority tasks. The most critical aspect is the main menu design, as its navigational capabilities and quick access to core features will determine whether users find the application useful. No matter how powerful the system is, if it's inconvenient to use, users won't engage with it.

4.1.4 Main Menu Design

From my analysis of LazyDocker in Section 2.4.1, I identified several features I would like to include in my solution.

Here, I adapt them to suit the needs of my utility:

- **Panes**: Kafka data is naturally suited for display in table format, as most elements can be easily categorized.

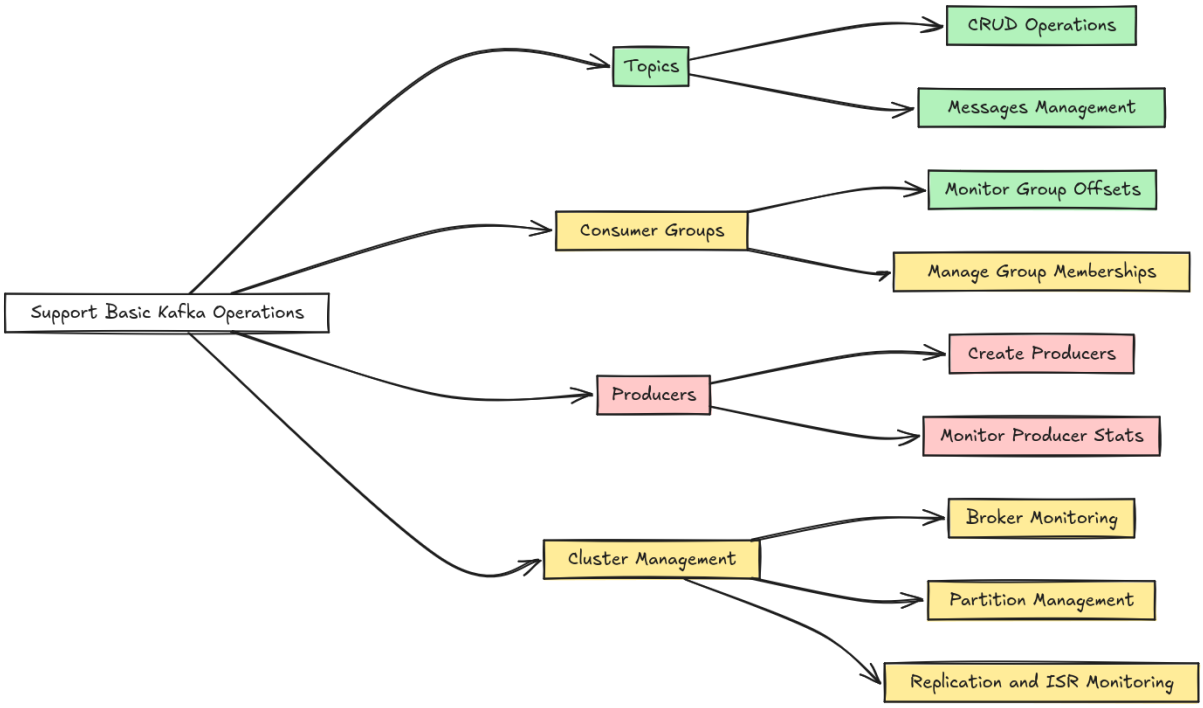


Figure 4.4: Basic Kafka operations requirements

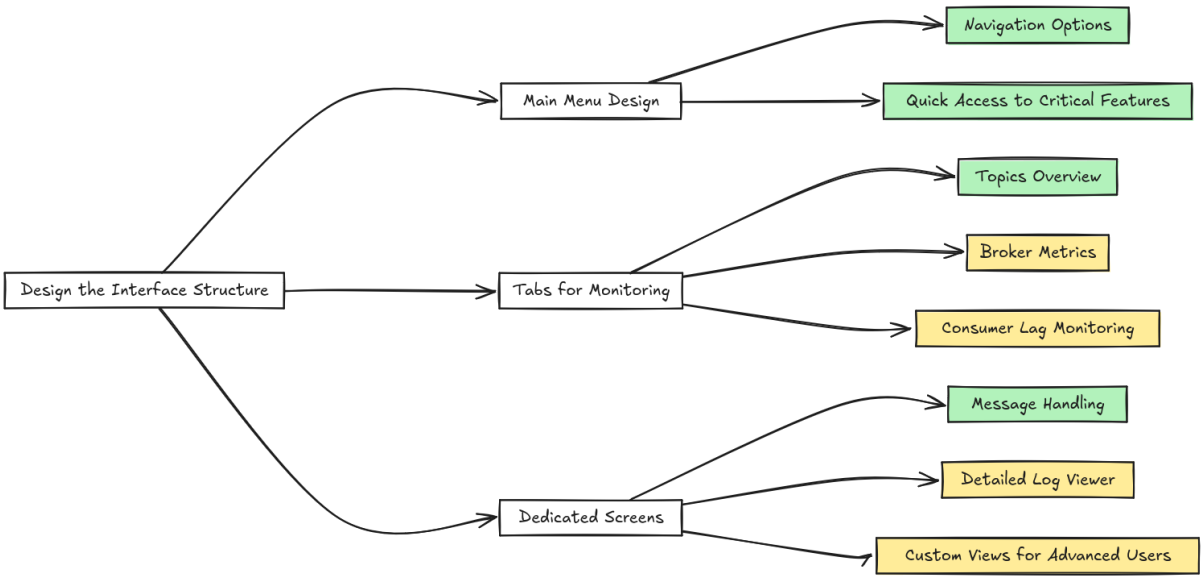


Figure 4.5: Interface design structure

- **Hotkeys**: Hotkeys are what make navigation satisfying. They are essential for any TUI utility. I plan to implement a general hotkeys bar and context-specific menus for individual panes. For example, sorting topics by different parameters can be performed using hotkeys.
- **Navigation**: Navigation is equally important. Moving between panes should be intuitive and straightforward. I decided to support navigation not only with `Tab`/`Shift+Tab` and arrow keys but also with vim-style hotkeys. Many developers use vim/emacs plugins in their workflows, and it's enjoyable to have familiar keybindings across the developer environment.
- **Colors**: As mentioned in the analysis, colors play both an aesthetic and functional role. They help highlight important elements and focus the user's attention where needed. Ideally, I would like to provide customizable palettes, but for the MVP, I will hardcode a basic color palette. If time permits, I will later update it to be configurable.

Inspired by LazyDocker, I sketched the following main menu mockup:

Figure 4.6: Main menu mockup

The first MVP version will only include the topic view and message view. I will discuss these in detail below. For now, a brief note on consumer and broker views: when designing mockups, I drew inspiration from Kafka Manager as a free alternative to Conduktor (which was discussed by Sarah). All views include a data table with various columns. For consumer groups, an initial implementation could include only group names (IDs) and associated topics. The broker screen is intended to track load metrics, so I divided it into two panels—a table panel with general information about each broker and a detailed information panel for the selected broker. In the future, the second panel could include tabs, such as one for rendering load graphs or displaying data loss statistics. To determine their utility, I will consult with my stakeholder.

Topics View: This is the main screen users will interact with most frequently, so it will be the first to be implemented for the MVP. This screen is divided into three panes:

- A general pane displaying information about topics in the cluster.

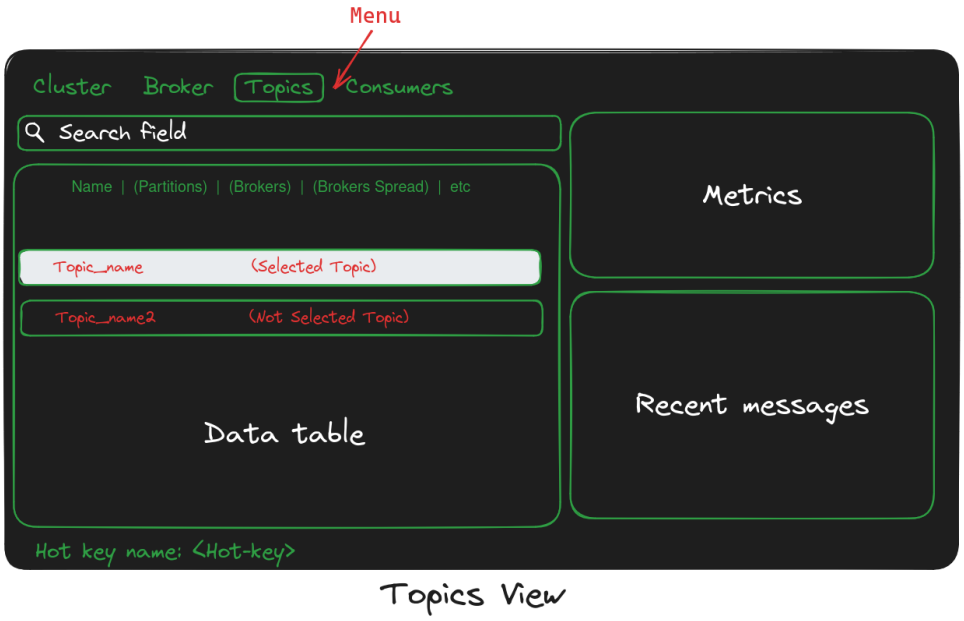


Figure 4.7: Topics screen mockup

- A metrics pane for the selected topic, potentially including graphs for specific time periods. Initially, it will display basic data.
- A messages pane showing the last *N* messages in the selected topic. By pressing enter on any topc will open new screen with messages stored in this topic (Figure **??**).

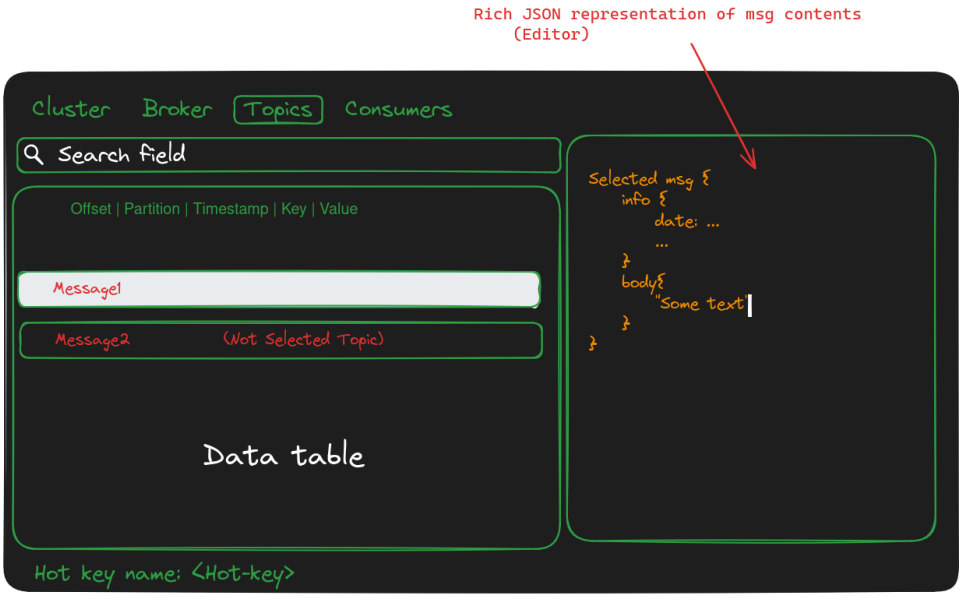


Figure 4.8: Message view

Overall, this screen is not drastically different from the topics screen. It presents messages in a tabular format, and when a message is selected, its content can be viewed in an editor displayed on the right side of the screen.

4.1.5 Tech Stack

Having finalized the two most critical parts of the interface, I can now determine the technology stack. Below, in Figure 4.9, I have divided the technologies into three categories.

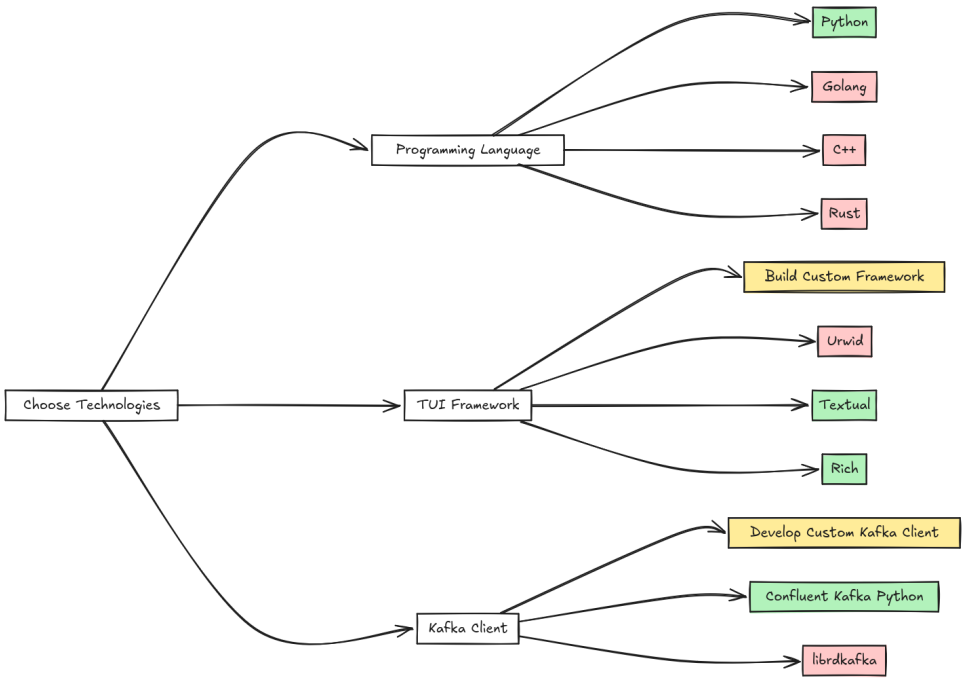


Figure 4.9: Technology Stack Overview

Programming Language: Choosing the programming language is a key decision for the project, as it affects performance, coding style, and the overall development approach. I considered using Rust or Go, but I am not sufficiently experienced with these languages to confidently complete the project within the given timeframe. C++ and C were also on the rejected list for the same reason, even though they are promising choices for creating lightweight, optimized interfaces for Linux command-line environments. These languages, in my opinion, would be excellent for developing a lightweight and optimized terminal interface tailored for Linux users. However, I chose Python instead. Python is both simple to read and powerful, with an extensive library ecosystem. Additionally, I have been working with Python since I was 8 years old, which enables me to estimate tasks with a high degree of accuracy in terms of time. For these reasons, Python will form the foundation of this project.

TUI Framework: The next most important aspect is the user interface. After conducting research, I identified four options. `Urwid` is a relatively outdated library with sparse documentation, but from the screenshots, it seems to include all the essential components for creating an interface based on my mockups. Its downsides are that it only supports a few terminal emulators and does not run on Windows at all. Then, I discovered `Textual` and was immediately impressed with its documentation—everything is well-documented, and the library includes numerous widgets required for developing a terminal interface. Additionally, `Textual` claims to be nearly universal, running seamlessly across different environments while maintaining consistent appearance, which is crucial for a good TUI utility. You may notice that I marked not only `Textual` as my preferred framework but also another option—`Rich`. Technically, `Rich` was created by the authors of `Textual` and integrates fully with it, so I consider them as a single framework, even though they are technically distinct libraries.

I also highlighted the option of building a custom framework in yellow. This option is highly unlikely but not impossible. If `Textual` fails to meet my needs, I may have to explore other options or develop a solution myself, which would require significantly more time. However, within the scope of this coursework, it is unlikely that I will pursue this route.

Kafka Client: The wrapper cannot function without connecting to the system itself. I found two options, but `confluent_kafka` has better and simpler documentation, along with a more active GitHub repository. Therefore, I chose this library. As with the TUI framework, creating a custom framework is always an option, but it is an extremely complex task and is unlikely to be part of this coursework.

4.1.6 Testing and Debugging

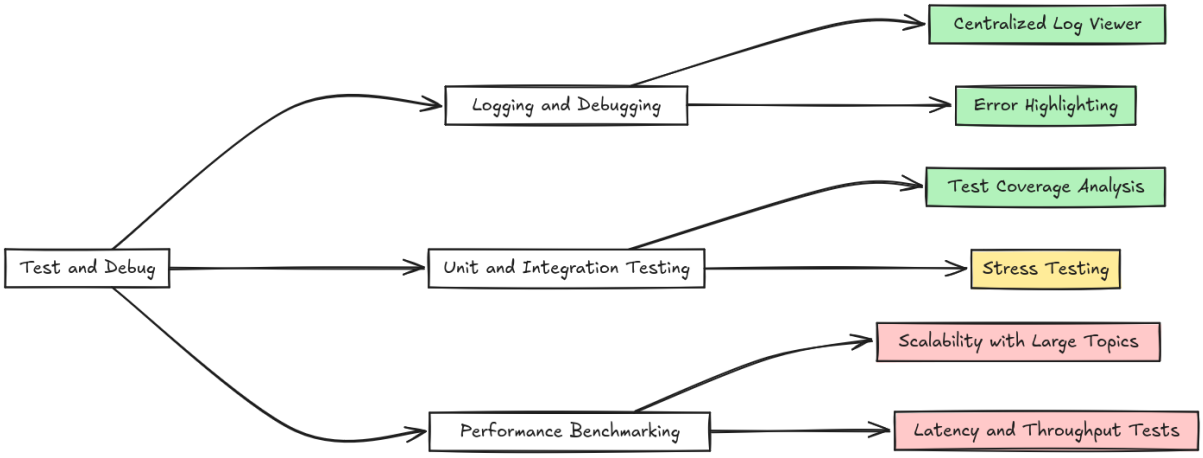


Figure 4.10: Testing and Debugging Plan

It is crucial to establish the testing boundaries early on to avoid spending excessive time on non-critical areas. Logging is an essential feature for any project, as debugging without logs can become a nightmare. The same applies to error highlighting—errors should be highlighted in the logs to make them easier to locate in large log files. For the evaluation phase, I need to provide proof of testing, and since I plan to use a Test-Driven Development (TDD) approach during development, tracking the percentage of code covered by tests will be helpful.

Since this is an MVP, stress testing and scalability are not critical at this stage. The MVP's goal is to function and demonstrate the system's capabilities to the stakeholders. Like a sketch in drawing, an MVP provides sufficient understanding but is almost never the final extensible solution. Therefore, I decided not to include performance benchmarking in the initial releases, and stress testing will only be conducted if time allows.

4.1.7 Release and Support

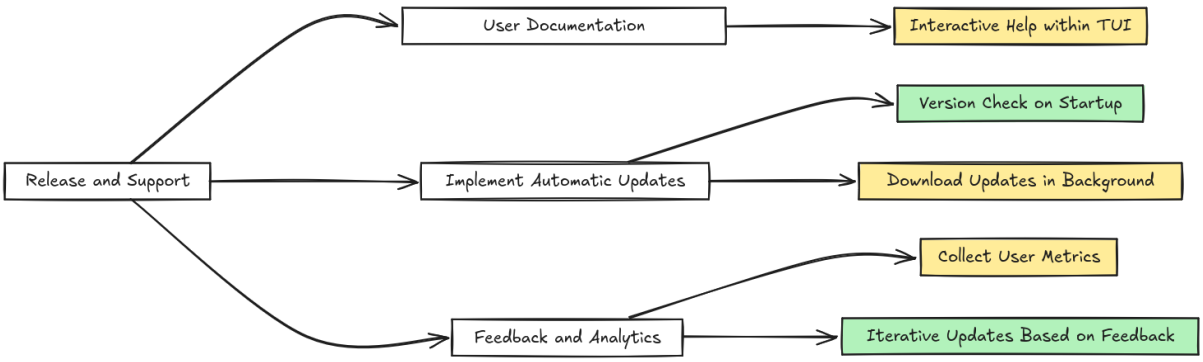


Figure 4.11: Release and Support Plan

Supporting the product during and after development is as important as the development itself. I divided this section into three parts: documentation, updates, and feedback/analytics.

Documentation: Documentation is a critical part of any project involving multiple developers. However, since this is a school project with limited time, and good documentation is typically time-intensive, I will focus on inline code comments and Python's built-in `pydoc` library (https://docs.python.org/3/library/pydoc.html), which converts comments in the code into well-formatted HTML and CLI manual pages.

Updates: Tracking application versions is good practice. In the final application, I could offer users the option to update the package when new versions are released. However, this task is not essential for the MVP, and I find it challenging to estimate the time required for implementing this functionality. Thus, this task will be deferred until after the core features of the first release are completed.

Feedback and Analytics: These are two things that help improve the application. In this project, I work iteratively with my stakeholders, collecting feedback to refine the application. I also believe that assessing an application's performance requires more than just feedback—quantifiable metrics are needed for a more accurate evaluation. However, collecting metrics from the beginning, before acquiring any users, is relatively pointless and is not critical for developing a good MVP. Therefore, assuming a small number of initial users (stakeholders), I plan to implement this feature in future releases.

4.2 Structure of the Solution

We have already discussed the interface layout in Section 4.1.4, but now, based on the technical decisions—particularly the choice of programming language—we can consider the design of the backend, which is the core of the application responsible for processing and transferring data from Kafka to the frontend.

In real-world development, the Domain-Driven Design (DDD) approach is widely used. DDD not only helps structure the project but also facilitates communication between technical teams and stakeholders. My stakeholders are developers, so they can generally understand technical discussions. However, using DDD will allow us to "speak the same language." For example, instead of saying, "That thing in Kafka that stores messages," I can simply say "Topic," and everyone will understand. This is the main advantage of DDD—it aligns terminology and simplifies collaboration. Additionally, DDD makes it easier to structure the project architecture.

To understand the subsequent sections, it is essential to define what a design pattern is. A **design pattern** is a frequently used solution to a common problem encountered during software architecture design. Design patterns serve as guidelines for structuring your code in a way that promotes flexibility, reusability, and scalability. To explain it in simple terms, a design pattern is like a blueprint or template for solving a specific problem. For further details, see https://refactoring.guru/design-patterns/what-is-pattern.

For the application, I decided to base the architecture on the MVC (Model-View-Controller) design pattern. MVC is a popular method of organizing code. The central idea of MVC is that each section of your code has a distinct purpose:

- **Model**: Handles the data and business logic of the application. It fetches and processes data, typically from Kafka in our case.
- **View**: Manages the display and presentation of the data. For this project, it will be responsible for the terminal user interface (TUI).
- **Controller**: Acts as the intermediary between the Model and View. It processes user input, updates the Model, and refreshes the View accordingly.

The primary advantage of MVC is that it separates concerns, making the code easier to maintain and scale. Moreover, since I am following a Test-Driven Development (TDD) approach, MVC naturally supports testing by isolating the logic in the Model and keeping the View and Controller relatively lightweight.

4.2.1 MVC use

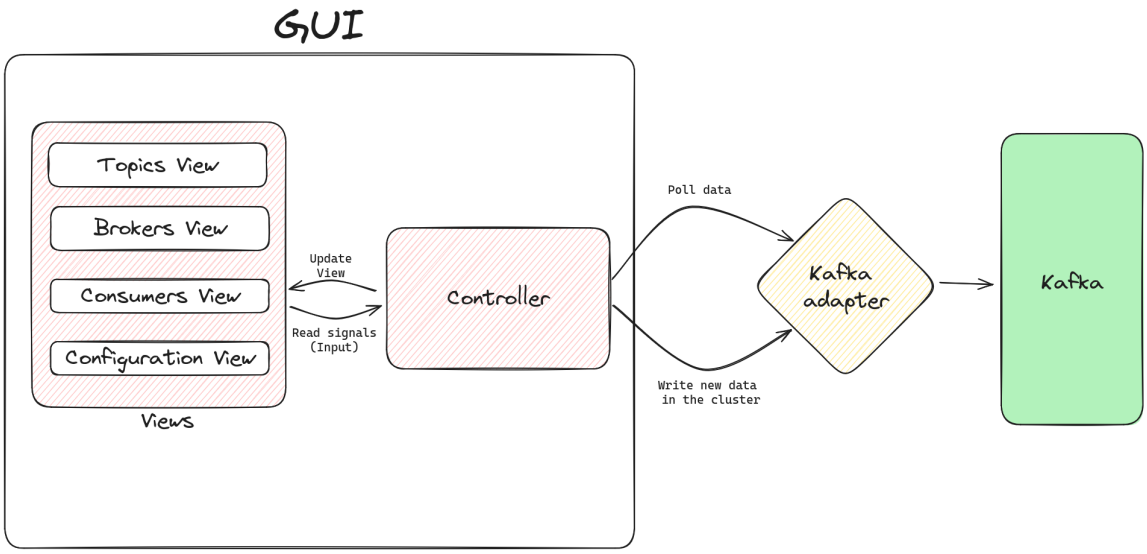


Figure 4.12

In Figure 4.12, I present a high-level overview of how I plan to implement the MVC pattern for this project. Let us briefly examine each component:

Model

The Model will handle all interactions with Kafka. This includes:

- Fetching metadata (e.g., list of topics, consumer groups).
- Reading and writing messages to and from topics.
- Monitoring cluster health and performance metrics.

View

The View is responsible for rendering the user interface. I will use the `textual` and `rich` libraries to build an intuitive terminal-based UI. The main components include:

- **Main Menu**: Displays the core features and navigational options.
- **Data Tables**: For presenting topics, messages, and metrics in an organized format.
- **Message Viewer**: Provides a detailed view of individual messages, basically editor.

Controller

The Controller will handle user inputs (e.g., keyboard shortcuts, navigation commands) and translate them into actions. For example:

- If a user selects a topic, the Controller will fetch the corresponding messages via the Model and update the View.
- If a user deletes a topic, the Controller will invoke the appropriate method in the Model and refresh the View.

By centralizing input handling in the Controller, we ensure that the business logic in the Model remains decoupled from the presentation logic in the View.

4.2.2 Architecture

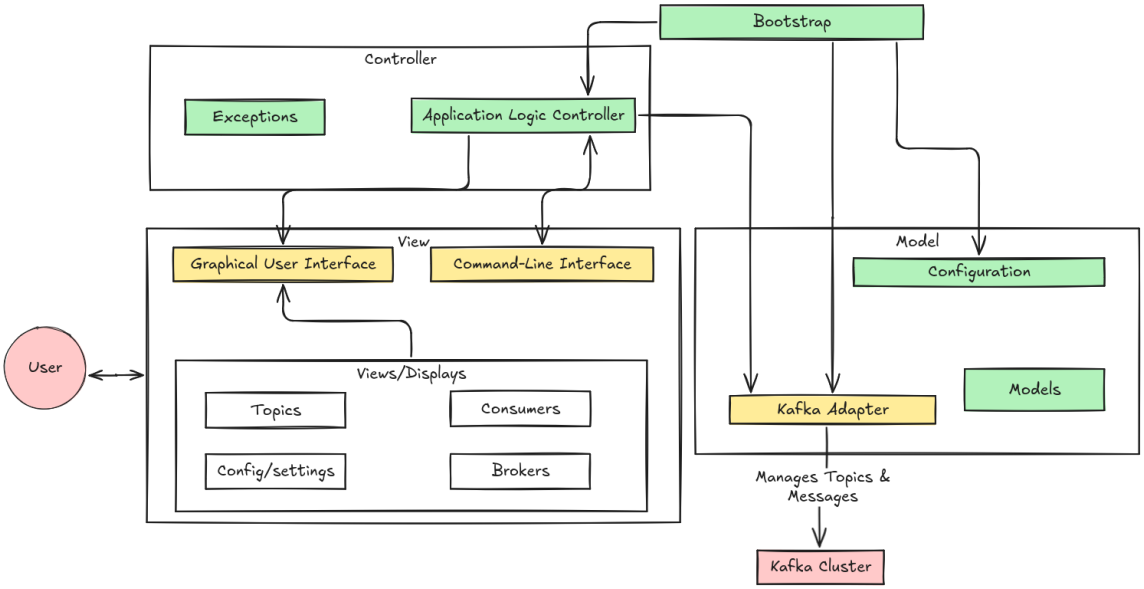


Figure 4.13: Detailed Architecture Overview

After spending some time sketching out ideas, I arrived at this architecture. All components are divided into three categories and follow the MVC pattern. The system is initialized from a bootstrap class, and public objects are stored in this class to allow them to be accessed from anywhere within the system.

- **Controller**: The brains of the application, responsible for processing and dispatching signals to other systems. Includes:
  - **Exceptions**: A set of custom exceptions that the system will throw in case of errors. While not critical for an MVP, they will enhance user feedback when reporting bugs and speed up development by clarifying error-handling logic.
- **Model**: Holds all the data and the classes needed to access this data. Includes:
  - **Configuration Class**: A class that stores configurable parameters. During initialization, it loads data like the Kafka host, login, and password from configuration files or environment variables. Centralizing all configurable parameters makes refactoring simpler by clearly separating responsibilities
  - **Models**: Specifications for the data models the system works with. Essentially, these are data classes used to validate incoming and outgoing data
  - **Adapters**: Adapters follow a design pattern and act as an interface to external systems. Multiple adapters could be supported, but for the MVP, I plan to focus on Kafka. The structure of the Kafka adapter is detailed in subsection 4.2.4
- **View**: The frontend of the application, containing the views or displays that the user interacts with. The controller reads the user input, processes it, and displays the appropriate information through the views. There could be multiple views (e.g., web, CLI, native app), each with a different implementation but consistent functionality. I plan to implement in the first release a Topics view (TUI) and configureable parameter inputs (CLI)

4.2.3 Models

In the first release, I will focus on Topics and Messages. Therefore, the primary models will be `Topic` and `Message`. However, given that Kafka topics are divided into partitions, and messages reside in partitions rather than directly in topics, it makes sense to add `Partition` as a model. This simplifies data parsing and supports future functionality.

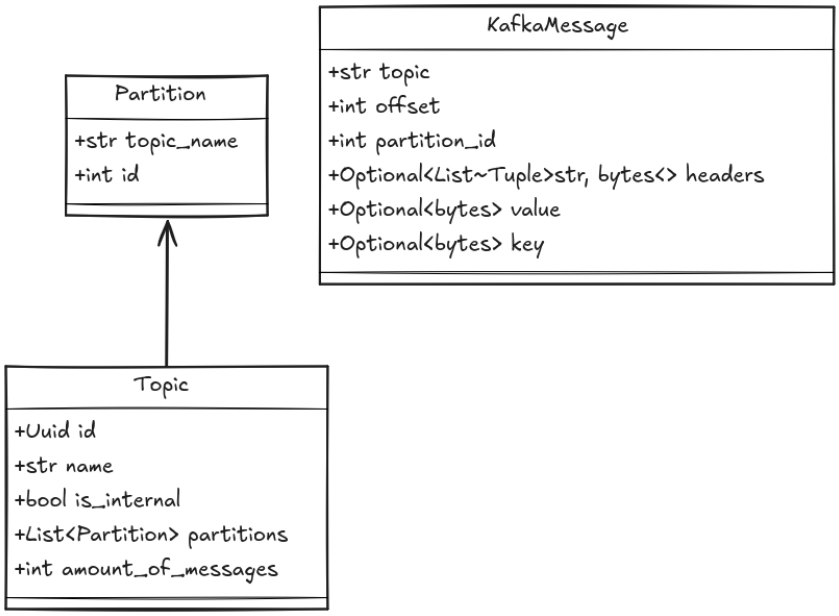


Figure 4.14: Model Structure

The models are based on Kafka objects and initially use Kafka's native data types to avoid unnecessary data conversion overhead.

4.2.4 Adapter

An adapter is an interface, and every interface has a specification. For example, an adapter converting from a UK plug to a European socket must always take three prongs as input and produce two as output. Regardless of the internal implementation of the adapter it follows this ”rules”. Similarly, here, I use an abstract class to specify the adapter interface for Kafka.

The abstract class defines the abstract methods that the business logic will interact with. An adapter class will inherit from this abstract class and provide concrete implementations of these methods. This concept is easier to understand when viewed in Figure 4.15.

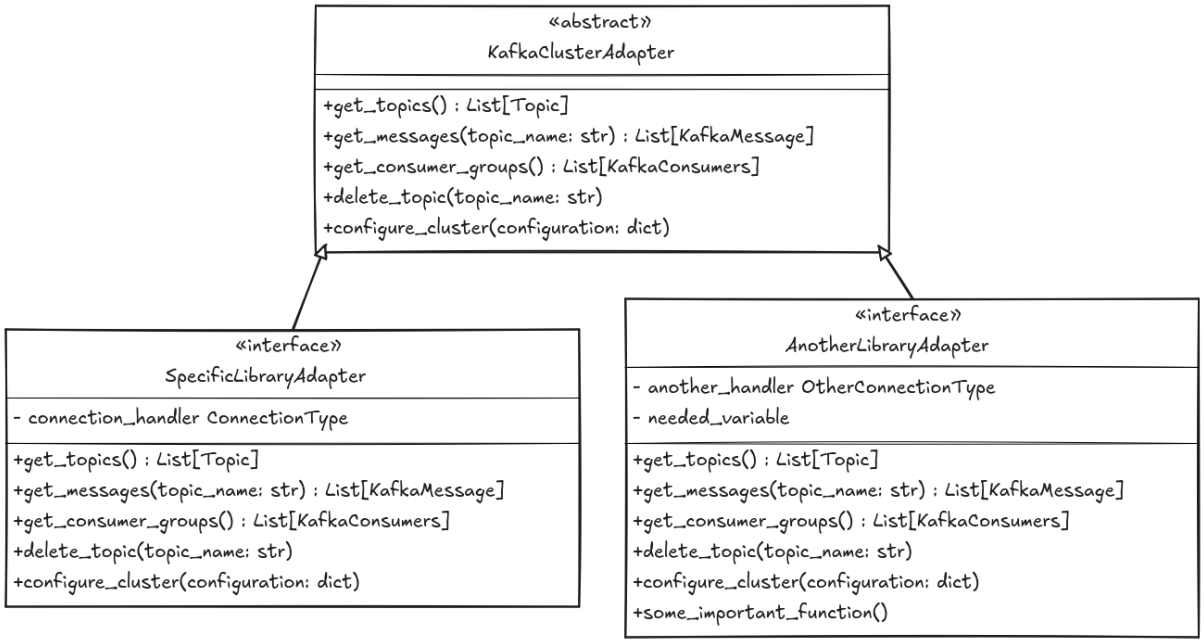


Figure 4.15: Adapter Example

I have outlined several potential methods for the system, but for the first release, I will focus on implementing only two: `get_topics()` and `get_messages()`.

5 Development

5.1 Preparation

5.1.1 Folder Structure

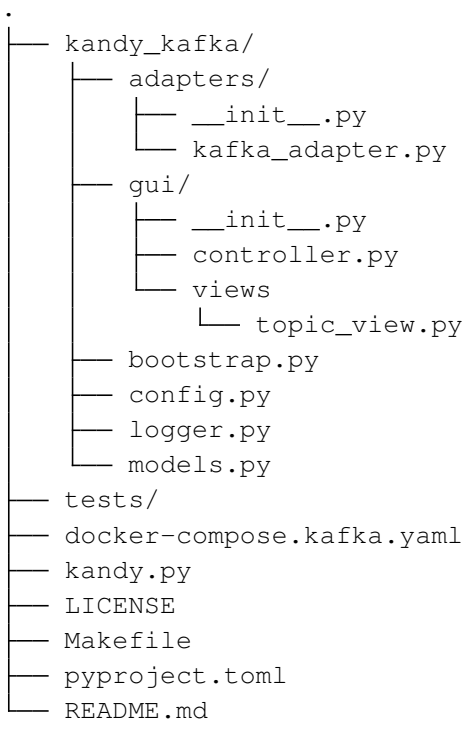


Figure 5.1: Project folder structure

For this project, I chose to use the 'poetry' dependency manager instead of the standard 'pip'. 'Poetry' simplifies the management of environments and dependencies but requires a specific project structure. For instance, the code must reside in a folder named after the project unless specified otherwise in the configuration file. Additionally, the root directory must contain a `README.md` file, along with all Git-related files and scripts (e.g., test scripts).

The core code is stored in the `kandy_kafka/` folder. Below, I describe the purpose of each file and folder:

- **README.md**: Contains project information, including a description, requirements, and installation instructions.
- **pyproject.toml**: Used by `poetry` to define dependencies and project metadata. A detailed version is available in Appendix B.1.
- **Makefile**: A script for automating tasks using the `make` tool[1].
- **LICENSE**: Since I plan to open-source this project after my A-levels, I included the GPL license to guarantee end-user freedoms to run, study, share, and modify the software.

[1] **Make** is a command-line tool for executing tasks defined in a configuration file.

- **kandy.py**: The entry point of the application that the user will run.
- **docker-compose.kafka.yaml**: This file specifies the configuration for Docker Compose, which I use to set up a Kafka environment for testing.
- **kandy_kafka/**: Contains the core project code:
  - **adapters/**: Holds adapter and repository files.
  - **gui/**: Contains views and the controller.
  - **bootstrap.py**: Defines the application startup logic.
  - **config.py**: Handles external configuration parameters.
  - **logger.py**: Manages logging setup using Python's built-in `logging` library.
  - **models.py**: Contains data models, represented as Python `dataclasses`.
- **tests/**: Directory containing automated tests.

This folder structure enforces separation of concerns, helping to write cleaner, maintainable code.

5.1.2 Libraries

The latest version of `pyproject.toml` is provided in Appendix B.1. For now, I have included the following libraries:

- **pytest**: A testing framework for creating and running test cases efficiently.
- **pydantic**: A data validation library
- **confluent-kafka**: A library chosen during the design phase to handle Kafka cluster communication.
- **urwid**: Although I initially selected `textual` as the primary library for GUI development, I decided to prototype one view using `urwid` for comparison and learning purposes. This experiment will help refine my choice.

5.2 Coding

5.2.1 Project Skeleton

I began by creating the skeleton of my application. This involves defining the classes and functions I plan to use, without implementing the actual logic yet.

```python
# kandy_kafka/logger.py

import logging


def setup_logger(log_level: str):
    """
    Set up the logger for the application.
    """
    # Remove all handlers associated with the root logger object.
    for handler in logging.root.handlers[:]:
        logging.root.removeHandler(handler)

    logging.basicConfig(
        format="%(asctime)s,%(msecs)d %(levelname)-8s [%(filename)s:%(lineno)d]
        ↪  %(message)s",
        datefmt="%Y-%m-%d:%H:%M:%S",
        level=log_level,
        filename="kandy_kafka.log",
    )
```

Listing 1: Logger setup function

For logging, I reused a function from one of my previous projects, making minor adjustments. The function initializes a logger and formats the log output to be written to a file with name `kandy_kafka.log` instead of `stdout`.

```python
# kandy_kafka/config.py

import os
from kandy_kafka import logger


class Config:
    LOGGING_LEVEL: str

    KAFKA_HOST: str
    KAFKA_PORT: int

    def __init__(self) -> None:
        self.LOGGING_LEVEL = os.getenv("LOGGING_LEVEL", "INFO")
        logger.setup_logger(self.LOGGING_LEVEL)

        self.KAFKA_HOST = os.getenv("KAFKA_HOST", "localhost")
        self.KAFKA_PORT = int(os.getenv("KAFKA_PORT", 9092))
```

Listing 2: Configuration class

Next on the line was configuration. This is a class with public fields that are filled from environment variables or configuration files at application startup. The configuration contains fields for the Kafka cluster host and port, which will later be used in the Kafka adapter. I also included the logging level in the configuration, allowing the application to adjust the detail level of logs (e.g., `ERROR`, `DEBUG`, `INFO`). Not necessary for this type of project but still I think it is a good practice.

```python
# kandy_kafka/adapters/kafka_adapter.py

from abc import ABC, abstractmethod


class AbstractKafkaClusterAdapter(ABC):
    """
    Abstract base class for Kafka cluster adapters.

    Defines the interface for interacting with a Kafka cluster, including fetching
    ↪ topics
    and retrieving messages from a given topic.
    """
    @abstractmethod
    def get_topics(self):
        raise NotImplementedError

    @abstractmethod
    def get_messages(self, topic_name: str):
        raise NotImplementedError


class KafkaAdapter(AbstractKafkaClusterAdapter):
    """
    Kafka adapter that implements the methods to interact with a Kafka cluster
    ↪ using
    confluent-kafka library.
    """
    def __init__(self, host: str, port: int):
        pass

    def get_topics(self):
        pass

    def get_messages(self, topic_name: str):
        pass
```

Listing 3: Kafka adapter implementation

The adapter interface defines the methods required for interaction with a Kafka cluster, such as retrieving topic names and fetching messages from topics. The implementation of the `KafkaAdapter` class is currently a placeholder, as the exact details depend on the chosen library. The abstract class ensures that any child class implements all required methods by raising a `NotImplementedError` if a method is missing. The `KafkaAdapter` class includes an `__init__` method that takes the Kafka host and port as arguments, preparing it for future implementation.

```python
# kandy_kafka/bootstrap.py

from dataclasses import dataclass
from kandy_kafka.adapters.kafka_adapter import KafkaAdapter
from kandy_kafka.config import Config


@dataclass
class Bootstraped:
    kafka_adapter: KafkaAdapter
    config: Config


class Bootstrap:
    bootstraped: Bootstraped

    def __call__(self):
        config = Config()
        kafka_adapter = KafkaAdapter(config.KAFKA_HOST, config.KAFKA_PORT)
        bootstraped = Bootstraped(
            kafka_adapter=kafka_adapter,
            config=config
        )

        return bootstraped
```

Listing 4: Bootstrap implementation

Next, I created the bootstrap mechanism. This class initializes and orchestrates the application components, making them accessible through a single point. The `Bootstraped` dataclass contains the objects that need to be shared across the system, such as the Kafka adapter and configuration.

The `__call__` method initializes the system components and returns a `Bootstraped` object. This approach allows components to be accessed globally without passing them explicitly between functions, simplifying the code.

```python
from kandy_kafka.bootstrap import Bootstrap

module = Bootstrap.bootstraped.module_name
```

Listing 5: Accessing modules through the bootstrap

This design avoids excessive dependency injection, making the system easier to debug and maintain.

5.2.2 First Release

In my project, I decided to use the approaches of TDD (Test-Driven Development) and BDD (Behavior-Driven Development). First, I define features using the Gherkin syntax, then I write tests that describe the functionality I want to implement, and finally, I write the code that will pass these tests.

The first feature in my plan is related to connection configuration. I aim to simplify manual testing and provide a more comprehensive picture to my stakeholders during demos. By allowing changes to be made to the system from outside (e.g., through startup flags or a configuration file), I must strictly validate the data and throw meaningful errors if the user makes a mistake during input.

```
Feature: Read and validate host configuration

  Scenario: Error when no configuration file is found
    Given Configuration file is not present
    When system loads config
    Then application should prompt user to create or specify a
     ↪  configuration file

  Scenario Outline: Configuration file has syntax error
    Given Configuration file is present
    And Configuration file has <error_type> syntax error
    When system loads config
    Then application should show <error>

    Examples:
      | error_type   | error                   |
      | missing_host | "Host is missing"       |
      | missing_port | "Port is missing"       |
      | empty_file   | "Configuration file is empty" |

  Scenario: Configuration file is valid
    Given Configuration file is present
    And Configuration file has valid syntax
    When system loads config
    Then config should have valid connection details

  Scenario: Configuration file has invalid yaml syntax
    Given Configuration file is present
    And Configuration file has invalid yaml syntax
    When system loads config
    Then application should raise yaml error
```

How should you approach reading this? As mentioned earlier, this is a Gherkin script, a structured language specifically designed for Behavior-Driven Development (BDD). Its primary purpose is to describe system behavior in plain text, bridging the gap between technical developers and non-technical stakeholders.

The Gherkin script is organized into **features**, which represent high-level functionalities of the system. Each feature is further divided into **scenarios**, outlining specific use cases or tests related to that feature. The scenarios follow a clear and consistent structure:

- **Given**: Sets up the initial state or preconditions required for the test.
- **When**: Specifies the action or event that triggers the behavior.
- **Then**: Describes the expected outcome or result.

Consider the following scenario:

```
Scenario: Error when no configuration file is found
  Given Configuration file is not present
  When system loads config
  Then application should prompt user to create or specify a configuration
   ↪  file
```

This scenario describes the behavior of the system when no configuration file is found. Each line is self-explanatory, making it accessible to developers and stakeholders alike.

For repetitive tests involving different inputs and outputs, Gherkin provides **Scenario Outlines**. These serve as templates for multiple test cases. For example:

```
Scenario Outline: Configuration file has syntax error
  Given Configuration file is present
  And Configuration file has <error_type> syntax error
  When system loads config
  Then application should show <error>

  Examples:
    | error_type   | error                   |
    | missing_host | "Host is missing"       |
    | missing_port | "Port is missing"       |
    | empty_file   | "Configuration file is empty" |
```

Here, the `<error_type>` and `<error>` placeholders represent variables that are defined in the `Examples` section. This structure allows the same scenario to be tested with multiple conditions.

```
import pytest
from pytest_bdd import parsers, scenarios, given, when, then
import yaml

from kandy_kafka.config import Config
from kandy_kafka.exceptions import HostsFileHasWrongSyntax,
 ↪  HostsFileNotFound
from pathlib import Path

scenarios("../features/hosts.feature")


# Non-existing configuration file scenario
@pytest.fixture
```

```python
@given("Configuration file is not present")
def non_existing_config_file(tmp_path):
    return tmp_path / "hosts.yaml"


@pytest.fixture
@when("system loads config")
def config():
    return Config()


@then("application should prompt user to create or specify a configuration
↪  file")
def check_prompt_to_create_config(config, non_existing_config_file):
    with pytest.raises(HostsFileNotFound):
        config.hosts_file = non_existing_config_file
        config.load_hosts("default")


# Wrong syntax scenario
@pytest.fixture
@given("Configuration file is present")
def config_file(tmp_path):
    file = tmp_path / "hosts.yaml"
    file.touch()
    return file


@pytest.fixture
@given(
    parsers.parse("Configuration file has {error_type} syntax error"),
    target_fixture="config_file_with_wrong_syntax",
)
def config_file_with_wrong_syntax(config_file, error_type):
    assert config_file.exists()
    config_fixture = (
        Path("tests") / "fixtures" / "hosts" /
        ↪  f"wrong_syntax_{error_type}.yaml"
    )
    config_file.write_text(config_fixture.read_text())


@then(parsers.parse("application should show {error}"))
def check_promt_to_fix_syntax(config, error, config_file):
    print(error)  # TODO check that valid error message is in the stderr
    ↪  (Or stdout)
    with pytest.raises(HostsFileHasWrongSyntax):
        config.load_hosts("default", config_file)

# Correct syntax scenario
@pytest.fixture
```

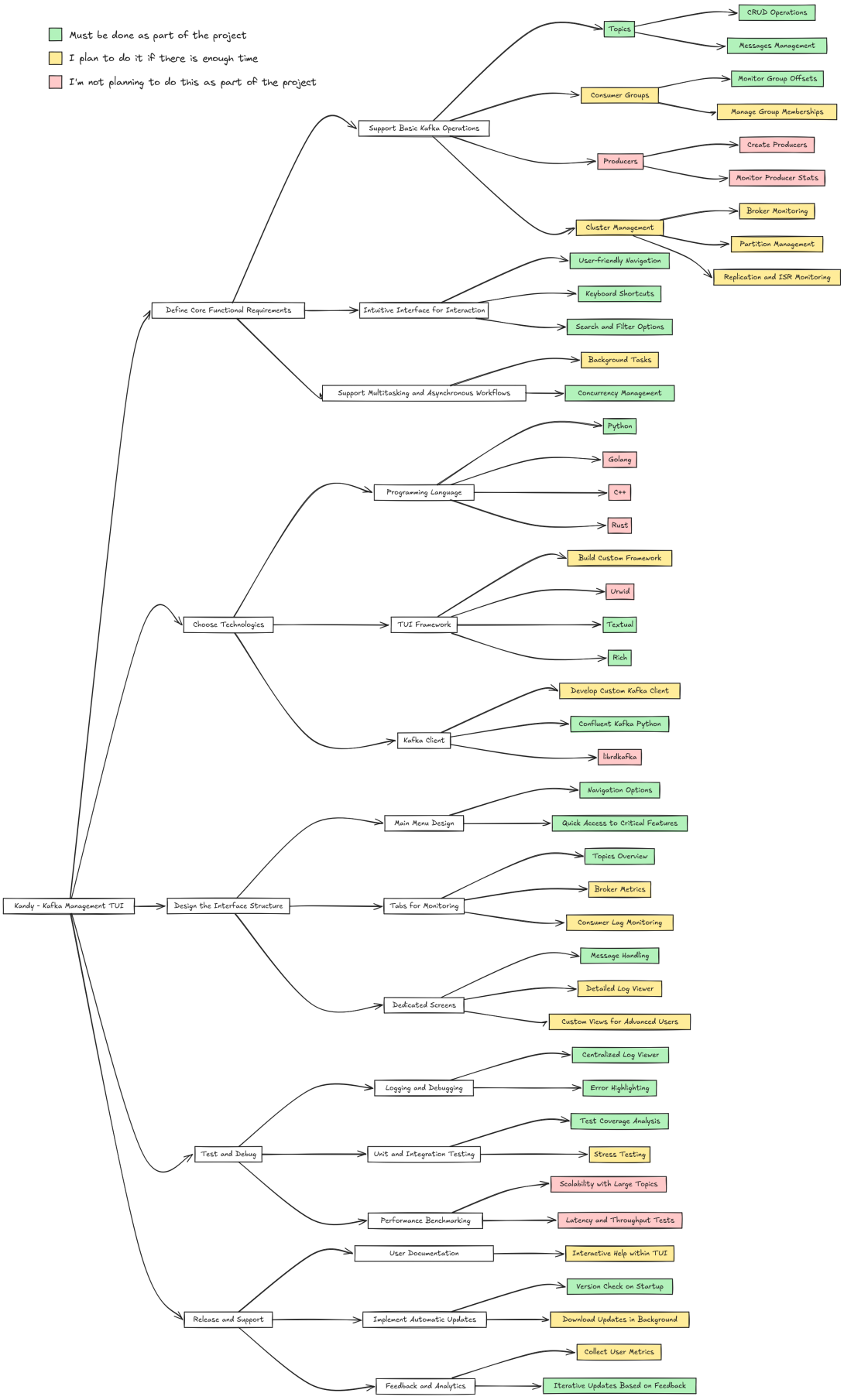
```
@given("Configuration file has valid syntax")
def config_file_with_correct_syntax(config_file):
    config_file.write_text(
        """
        default:
            host: localhost
            port: 9092
        """
    )


@then("config should have valid connection details")
def check_config_connection_details(config, config_file):
    config.load_hosts("default", config_file)
    assert config.KAFKA_HOST == "localhost"
    assert config.KAFKA_PORT == 9092


# Invalid yaml syntax scenario
@pytest.fixture
@given("Configuration file has invalid yaml syntax")
def config_file_with_invalid_yaml_syntax(config_file):
    config_file.write_text("][")


@then("application should raise yaml error")
def check_yaml_error(config, config_file):
    with pytest.raises(yaml.YAMLError):
        config.load_hosts("default", config_file)
```

The code implements the scenarios defined in the feature file using the `pytest-bdd` plugin.

In the BDD workflow, the **Given-When-Then** steps in the feature file correspond directly to Python functions in the test suite. These steps are implemented as pytest fixtures and assertions to validate the application's behavior. The `scenarios` function links the Python code to the feature file. This enables each scenario in the Gherkin script to execute its corresponding test implementation.

BDD Connection: Each **Given-When-Then** step in the Gherkin script is mirrored in the Python code using decorators: `@given`, `@when`, `@then`. For example, the "Missing Configuration File" scenario ensures that the application raises an error and prompts the user appropriately when no configuration file exists. Similarly, the "Valid Configuration File" scenario confirms that the configuration is parsed correctly, and the connection details are set as expected.

A System Design


Must be done as part of the project
I plan to do it if there is enough time
I'm not planning to do this as part of the project
Kandy - Kafka Management TUI
Define Core Functional Requirements
Support Basic Kafka Operations
Topics
CRUD Operations
Messages Management
Consumer Groups
Monitor Group Offsets
Manage Group Memberships
Producers
Create Producers
Monitor Producer Stats
Cluster Management
Broker Monitoring
Partition Management
Replication and ISR Monitoring
Intuitive Interface for Interaction
User-friendly Navigation
Keyboard Shortcuts
Search and Filter Options
Support Multitasking and Asynchronous Workflows
Background Tasks
Concurrency Management
Choose Technologies
Programming Language
Python
Golang
C++
Rust
TUI Framework
Build Custom Framework
Urwid
Textual
Rich
Kafka Client
Develop Custom Kafka Client
Confluent Kafka Python
librdkafka
Design the Interface Structure
Main Menu Design
Navigation Options
Quick Access to Critical Features
Tabs for Monitoring
Topics Overview
Broker Metrics
Consumer Lag Monitoring
Dedicated Screens
Message Handling
Detailed Log Viewer
Custom Views for Advanced Users
Test and Debug
Logging and Debugging
Centralized Log Viewer
Error Highlighting
Unit and Integration Testing
Test Coverage Analysis
Stress Testing
Performance Benchmarking
Scalability with Large Topics
Latency and Throughput Tests
Release and Support
User Documentation
Interactive Help within TUI
Implement Automatic Updates
Version Check on Startup
Download Updates in Background
Feedback and Analytics
Collect User Metrics
Iterative Updates Based on Feedback

B Files

B.1 pyproject.toml (Libraries and dependencies)

```
[tool.poetry]
name = "kandy-kafka"
version = "0.0.1"
description = "Handy way to manage kafka"
authors = ["Perchinka <alexsator.lukin@gmail.com>"]
license = "GPL-2.0-only"
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.11"
confluent-kafka = "^2.3.0"
urwid = "^2.6.10"
panwid = "^0.3.5"
pydantic = "^2.6.4"
pyyaml = "^6.0.1"
textual = "^0.81.0"

[tool.poetry.dev-dependencies]
pytest = "^8.1.1"
pytest-bdd = "^7.1.2"
pytest-cov = "^5.0.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Listing 6: pyproject.toml

References

[1] None yet