Phoenix
Project - 3
Team - 20

Adyansh Kakran	Meghana Tedla	Prakhar Jain	Rudra Dhar
2021111020	2021102002	2022121008	2022801002

Shrikara Arun
2021101058

1 Introduction

The Phoenix Auto Backup System addresses the critical need for automated and reliable file backup solutions for both individual users and organizations. Often, users face the risk of losing important data due to hardware failures, accidental deletions, or other unforeseen circumstances. This project aims to provide a comprehensive backup solution that automatically tracks file changes or performs periodic backups, ensuring data integrity and accessibility.
Note: For the rest of this report, whenever we refer to an *interface*, it is implemented as a specialization of the *abstract base class* provided by the `abc` package since python does not have a native interface system. The *delete* event is only triggered when a file is permanently deleted with `shift + del`

2 Requirements and Subsystems

2.1 Functional and Non-functional Requirements

2.1.1 Functional Requirements

- Data Compression: Automatically compress files prior to backup to reduce storage needs.
- Data Encryption: Encrypt files using a secure method before they are stored to ensure data confidentiality.
- Key Management: Manage encryption keys securely, including their generation, storage, and retirement.
- Support for multiple compression methods: Support different compression algorithms to account for different file types and compression needs.
- Support for multiple encryption methods: Offer multiple encryption protocols to accommodate different security needs and regulatory requirements.

2.1.2 Non-functional Requirements

- Security: Use proven cryptographic standards to ensure the security of data.
- Configurability: Allow for customization of encryption and compression settings based on user preference or policy.
- Efficiency: Compression and encryption should be fast and efficient to minimize backup time.
- Modularity: Design of the compression and encryption services should be modular, allowing for easy updates or replacements of algorithms as better techniques become available.
- Extensibility: The system should support easy integration of additional functionalities or components without significant redesign.

- Response Time: System response time for encryption, compression, and other operations should be within acceptable limits, ensuring a smooth user experience.

2.2 Subsystem Overviews

2.2.1 Observation Subsystem

Observation

- A strategy design pattern is used to switch between observation algorithms. The `ObservationStrategy` interface is to be implemented by observation strategies. It contains `start` and `stop` methods and requires a `Path` object for the directory being observed.
- The `EventDrivenStrategy` and `PeriodicStrategy` are the concrete strategies for `ObservationStrategy`.
- The `EventDrivenStrategy` observes the directories of interest for any event (the events defined are `created`, `modified`, `deleted`, `moved`). Upon an event, the strategy triggers a call to the event handler which will initiate the file backup. It uses `inotify` API for monitoring file system events.
- The `PeriodicStrategy` periodically triggers a call to the event handler for file backup.

Event Handler

- The `EventHandler` class is responsible for handling the events triggered by the observation strategies. It initiates the backup process for the files that have been changed. Depending on the type of event received, it either initiates the backup for a single file or for an entire directory.
- It gives information about the number of files being backed up and the size of the files using `WatchDirComposite` class.

Watch Directory Structure

- A composite pattern is used to get information regarding the number of files within a directory and the size of the files.
- The `WatchDirComponent` is to be implemented by both the composite `WatchDirComposite` and the children `WatchDirLeaf`.
- The `WatchDirComposite` class is the container that comprises any number of files, including other directories. A `WatchDirLeaf` class is used to represent a file.
- A `WatchDirComposite` class has the same methods as a `WatchDirLeaf` class. However, instead of doing something on its own, it passes the request recursively to all its children and "sums up" the result.

2.2.2 Compression and Encryption

Compression

- A strategy design pattern is used to switch between compression algorithms. The `TarCompressionStrategy` class defines the concrete strategy. A `NoCompressionStrategy` class is also provided, but should not be used when backing up folders since we abstract folders into a compressed file from this stage on which the `NoCompressionStrategy` does not use.

- The `ICompressionStrategy` interface is to be implemented by all compression strategies. It contains `compress` and `decompress` functions along with an initalization for the output directories for each. The default output structure is:

```
.phnx
|-- compressed
|-- decompressed
|-- decrypted
|__ encrypted
```

in the directory where the script is run.
> [!WARNING]
> When compressing a folder, use `folder_name, False` to indicate a folder.

Encryption

- A strategy design pattern is used to switch between encryption algorithms. The `IEncryptionStrategy` interface is to be implemented by encryption strategies. It contains `encrypt` and `decrypt` methods along with an initialization for output directories, as mentioned in the compression section.
- A singleton design pattern is used to generate and manage the encryption key, defined in `KeyManager` which uses `SingletonMeta` metaclass, an implementation of the singleton pattern. Encryption algorithms might have differing implementations for this, so we also introduce a strategy design pattern in `IKeyStrategy`.
- We use GPG to perform encryption. The concrete strategy for `IEncryptionStrategy` is `GPGEncryptionStrategy`. The concrete strategy for `IKeyStrategy` is `GPGKeyStrategy`. Also, we need to ensure a single `gpg` object is used across encryption and key management. A singleton `GPGSingleton.py` is used to initialize gpg with desired config.

Manager

The `CEManager` owns the compression and encryption strategy that will be used. It is the context class for the strategy pattern for both compression and encryption algorithms. It abstracts away the use of the key manager from the caller.

2.2.3 File Upload and Download

Upload/Download Strategy Pattern

- The `UploadDownloadStrategy` abstract class defines the interface for different upload and download strategies. It contains abstract methods `get_name`, `upload`, and `download`.
- Concrete strategies implement this interface, such as `OneDrive` and `GoogleDrive`.

Uploader

- The `Uploader` class is the context that uses the strategy pattern. It has a composition relationship with the `UploadDownloadStrategy`.
- The `Uploader` class provides methods `upload_file` and `download_file` which delegate the respective operations to the currently set strategy.
- The strategy can be changed at runtime using the `strategy` property setter.

OneDrive Strategy

- The `OneDrive` class implements the `UploadDownloadStrategy` interface for OneDrive upload and download operations.
- It utilizes the Microsoft Graph API and requires authentication credentials (`client_id`, `authority`) and configuration (`siteName`, `site`, `domain`, `onedrive_path`).
- The `upload` method uploads a local file to the specified OneDrive path, generating a timestamped filename and logging the operation in a CSV file.
- The `download` method downloads a file from the specified OneDrive path.

Google Drive Strategy

- The `GoogleDrive` class implements the `UploadDownloadStrategy` interface for Google Drive upload and download operations.
- It requires authentication credentials stored in the `credentials.json` file and a `folder_id` specifying the destination folder on Google Drive.
- The `upload` method uploads a local file to the specified Google Drive folder, logging the operation in a CSV file.

- The `download` method downloads a file from Google Drive using the provided `file_id`.

Usage

- Create an instance of the `Uploader` class with the desired strategy (e.g., `OneDrive` or `GoogleDrive`).
- Call the `upload_file` or `download_file` methods on the `Uploader` instance, providing the necessary file paths and arguments.
- The respective strategy will handle the upload or download operation.
- The strategy can be changed at runtime by setting the `strategy` property of the `Uploader` instance.

> **[WARNING]** > Make sure to provide valid credentials and configuration for the respective upload/download strategies.

3 Architecture Framework

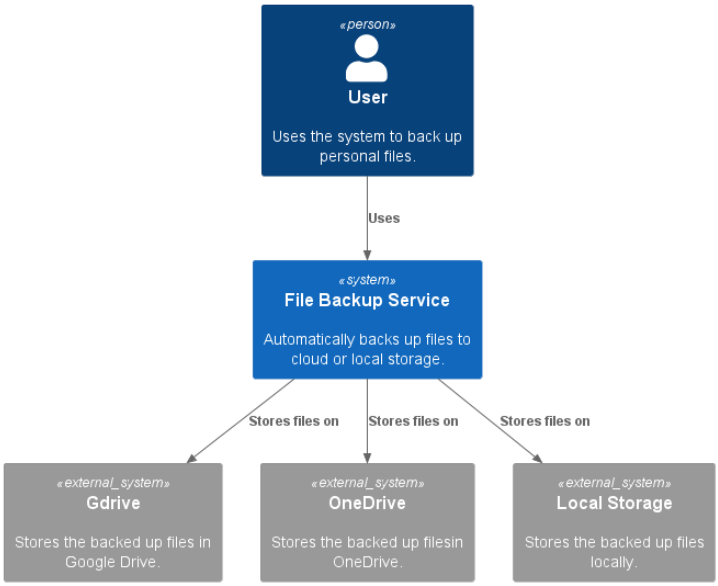

Figure 1: Context Diagram as per C4 model

3.1 Stakeholder Analysis (In accordance with IEEE 42010 Standard)

3.1.1 Stakeholder Identification

- **End Users**: These are the individuals who will actually use Phoenix to backup their files. They could be individuals or small businesses looking for a simple and reliable backup solution.
- **Developers**: These are the people who contribute to the development and maintenance of Phoenix. They need to ensure the system is robust, efficient, and meets the needs of the end users.
- **Cloud Engineers**: The cloud engineer is responsible for designing, implementing, and maintaining the cloud infrastructure where Phoenix will store its backup data.

3.1.2 Stakeholder Concerns

- **End Users**

  1. **Ease of Use**: End users may be concerned about the simplicity and intuitiveness of the Phoenix interface. They want to ensure that it's easy to configure backups, restore files when needed, and monitor the backup process.

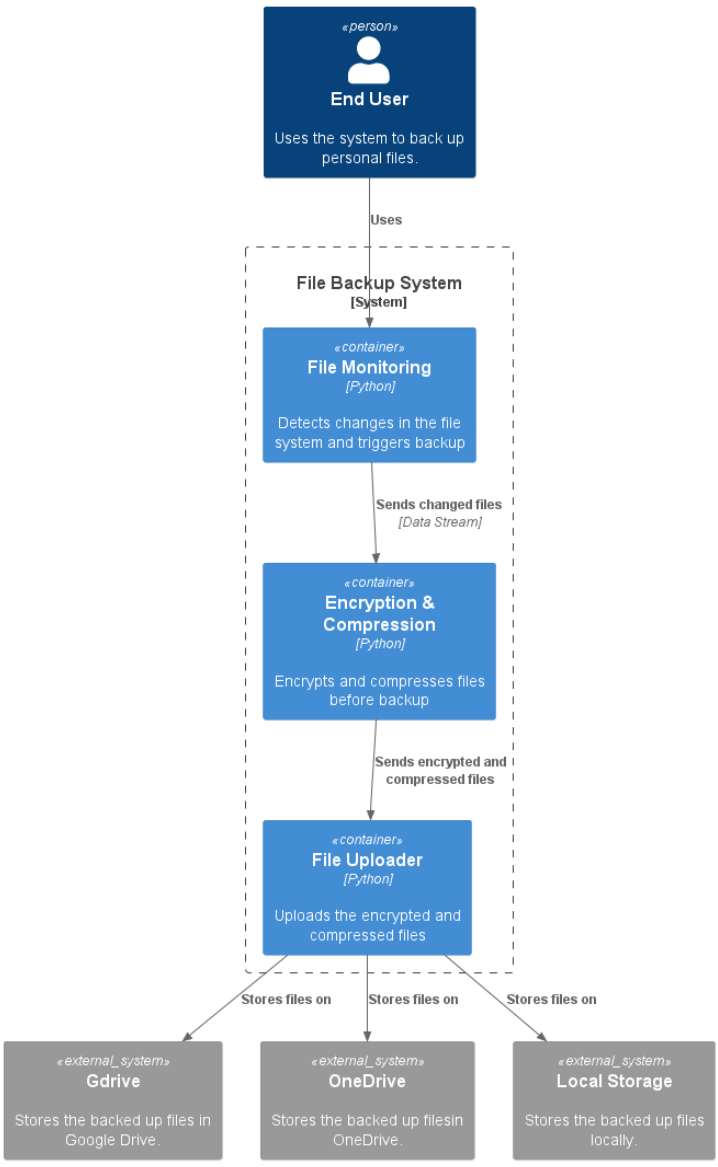

Figure 2: Container Diagram as per C4 model

2. **Data Security**: End users are concerned about the security of their backed-up data. They want assurance that their files are encrypted during transmission and storage and that access controls are in place to prevent unauthorized access.
3. **Reliability**: End users rely on Phoenix to securely and reliably back up their files. They are concerned about data integrity and want assurance that their files will not be corrupted or lost during the backup process.

- **Developers**

1. **Code Quality and Maintainability**: Developers are concerned about the quality and maintainability of the Phoenix codebase. They want assurance that the code follows best practices, is well-organized, and is easy to understand and modify.
2. **Extensibility**: Developers are concerned about the extensibility of the software, because they will be the ones adding new features as the product grows and starts becoming more complex.
3. **Testing**: Developers are concerned about the effectiveness of testing procedures, including unit tests, integration tests, and user acceptance tests, to ensure the reliability and stability of Phoenix.

- **Cloud Engineers**

1. **Data Privacy**: Cloud engineers are concerned about ensuring the privacy and confidentiality of backed-up data stored in the cloud, including compliance with data protection regulations and encryption of sensitive information.
2. **Scalability**: Cloud engineers need to ensure that the cloud infrastructure supporting Phoenix is scalable to accommodate growing storage requirements and increasing demand from users.
3. **Cost Management**: Cloud engineers are concerned about managing costs associated with cloud storage, including optimizing resource utilization, minimizing data transfer fees, and selecting cost-effective storage options.

3.1.3 Viewpoints and Views

- **User Experience Perspective**

  This viewpoint focuses on ensuring a positive and seamless user experience for end users interacting with Phoenix.

  Concerns addressed:

  1. Ease of use and intuitive interface design
  2. Accessibility considerations for users with disabilities
  3. Consistency in user interactions across different platforms and devices

  Views:

  1. **CLI Design View**:
     Describes the design principles and guidelines followed to create a lightweight and user-friendly CLI interface for Phoenix. Documents the organization of commands, options, and arguments to optimize usability and efficiency.
  2. **Usability Testing View**:
     Outlines the results of usability testing conducted with target users to evaluate the effectiveness of the CLI interface. Identifies areas for improvement based on user feedback and usability metrics.

- **Cloud Integration Perspective**

  This viewpoint focuses on the integration of Phoenix with cloud storage providers, addressing concerns related to data security, scalability, and performance in the cloud environment.

  Concerns addressed:

1. Security measures to protect backed-up data stored in the cloud from unauthorized access or data breaches.
2. Scalability of the cloud infrastructure to accommodate growing storage requirements and increasing demand from users.
3. Performance optimization for efficient data transfer and storage operations in the cloud environment.

Views:

1. **Cloud Provider Integration View**:
   Describes the integration of Phoenix with various cloud storage providers, including authentication mechanisms and API usage. Documents security measures such as data encryption, access controls, and compliance with industry regulations.
2. **Performance Optimization View**:
   Discusses techniques and best practices for optimizing performance in the cloud environment, including data caching, compression, and network optimization. Identifies potential bottlenecks and optimizations to improve backup speed and efficiency.

These viewpoints provide a structured approach to understanding and addressing the concerns of stakeholders involved in the Phoenix project, ensuring that all relevant perspectives are considered in its design, development, and deployment.

3.2 Major Design Decisions (ADRs)

3.2.1 Adoption of Monolithic Architecture for Phoenix

Status
Accepted
Context
We envision Phoenix, an automatic backup system that uploads files to a configurable secondary cloud based storage. The application runs on client machine(s) and deals with files/folders that are mounted to the machine. There is a need for efficiency and performance to minimize CPU taxation and network usage. It needs to be lightweight and performant in order to seem "invisible" and not impend the daily actions of the user. Our team is currently facing a tight schedule and is not familiar with the development of microservices based or event driven applications.
Alternatives Considered:

- Monolith
- Microservice
- Event Driven

Decision
In this context, we propose using a **monolithic architecture** by taking note of the following:
- Phoenix is a low level application and performance must be a top priority. It should use a bit of resources but ensure high throughput backup to the secondary storage - At this low level, it is hard to divide Phoenix into microservices or as an event driven architecture though they provide better extensibility in the long run since - there are a limited number of independent units Phoenix can be divided into - each independent unit does not use an independent database - the overhead of inter-service communication or the event bridge will significantly impact performance - The team is not well versed with developing microservices or event driven applications. In addition to the problems mentioned above regarding performance, it can also result in incorrect application of the pattern, can result in unintended technical debt, deployment and monitoring issues.
Consequences

- Simplicity in development and deployment (fast time to market) - The codebase is relatively smaller with lesser configuration required since components can communicate with each other easily. Since the team members are familiar with the architecture, it will also speed up operations and reduce errors.

- Performance - Monolithic applications are more performant and efficient for low level applications due to integration of components.
- Easier testing - The unified nature simplifies testing since there are no external dependencies or inter-service communication.
- Scalibility concerns - Scaling a monolith is harder since the entire application will have to be replicated, but since our current requirement is only a single instance per machine, this is not a concern at the moment.
- Limited flexibility and extensibility - Future modifications and extensions can be hard to implement due to the coupling between modules.

3.2.2 Strategy to Backup Files

Status
Accepted
Context
The Phoenix Auto Backup System is designed to address the critical need for automated and reliable file backup solutions. Users face various risks such as hardware failures, accidental deletions, or other unforeseen circumstances that may result in data loss. To ensure data integrity and accessibility, the system must provide a comprehensive backup solution that efficiently tracks file changes or performs periodic backups.
Alternatives Considered:

- Store Backed-Up Files Completely: This approach involves storing complete copies of files each time a backup is performed.
- Use Differential Storage with difflib in Python: This approach involves storing only the differences (diff) between the current version of files and the previous version, utilizing the difflib library in Python to generate and apply the differences.

Decision
In this context, we propose using the **Store Backed-Up Files Completely** approach for file backup due to the following reasons: - **Simplicity and Ease of Implementation**: Storing complete files simplifies the implementation of the backup system. It reduces the complexity of managing and applying diffs to reconstruct files, making the system more robust and easier to maintain.

- **Support for All File Types**: Storing complete files ensures that the backup system can handle all types of files effectively. While storing diffs might be suitable for text-based files, it may not be ideal for media files or other binary formats where the differentiated version could be an overhead rather than a space-saving measure.
- **Data Integrity and Accessibility**: Storing complete files ensures data integrity and accessibility. Users can easily retrieve complete, unaltered copies of their files without the need for complex reconstruction processes. This approach aligns with the primary goal of the backup system, which is to reliably safeguard user data against loss or corruption.
- **Extensibility**: Storing complete files provides a more extensible foundation for future enhancements and features. It allows for straightforward integration with additional backup strategies or optimizations without the constraints imposed by a diff-based approach.

Consequences

- **Space Efficiency**: Storing complete files may consume more storage space compared to storing only the differentials between file versions. However, given the decreasing cost of storage and the importance of data integrity, the trade-off is deemed acceptable for the intended functionality and user experience of the Phoenix Auto Backup System.
- **Performance**: The performance of backup operations, particularly in terms of storage and network bandwidth utilization, may be impacted by storing complete files. However, optimizations such as compression and incremental backups can mitigate these concerns to a certain extent.

- **Compatibility**: Storing complete files ensures compatibility with a wide range of backup tools and systems. It simplifies interoperability and data migration, allowing users to leverage the backup system across different platforms and environments without compatibility issues arising from diff-based storage mechanisms.

3.2.3 Adoption of GPG for encryption and Tar for compression for Phoenix

Status
Accepted
Context
Phoenix should support compressing and encrypting the files and folders it backs up to minimize disk and network utilization and ensure security since the data can be stored on a third party storage provider. In this context, we need to decide a mechanism to compress files and a mechanism to encrypt files. Given the complexity and the high risk associated with implementing custom encryption methods, and the availability of proven third-party solutions, we must choose between leveraging established third-party tools or developing in-house capabilities.
Alternatives Considered:

- own encryption scheme
- GPG for encryption
- Fernet for encryption
- AES for encryption
- own compression scheme
- Tar for compression
- zip for compression
- 7z for compression

Decision
In this context, we decide to use **GPG for encryption and Tar for compression** due to the following reasons: - **Complexity and Risk of Custom Solutions**: Developing custom encryption methods is highly risky due to the potential for security flaws and vulnerabilities. Encryption requires careful implementation and continuous maintenance to protect against evolving security threats. Using established, widely-reviewed protocols like those provided by GPG mitigates these risks significantly. - **Compliance and Trust**: Third-party tools like GPG are compliant with various regulatory standards and are trusted in the industry. This compliance is crucial for meeting legal and security standards. Tar and GPG are widely used in the *nix community.
- **Community Support and Updates**: Tools like GPG and Tar have large communities and are regularly updated. This ongoing support reduces the risk of the tools becoming obsolete and ensures that security vulnerabilities are addressed promptly.
- **Cost and Resource Efficiency**: Utilizing third-party tools reduces the need for extensive development and maintenance resources that would be required for custom solutions. This allows the team to focus on other critical aspects of the backup system.
Consequences
Advantages:

- **Security**: Leveraging well-established tools reduces the risk of security flaws.
- **Maintenance**: Relying on third-party solutions with active community support and updates ensures that the system remains secure and functional over time.
- **Compliance**: Using well-known, compliant tools helps in adhering to legal and regulatory requirements.

Disadvantages:

- **Dependence**: Dependency on external tools can pose risks if these tools are discontinued or if their licensing changes unfavorably.
- **Flexibility**: Using third-party tools may limit customization options compared to a custom solution.
- **Performance**: There may be some performance trade-offs, as third-party tools might not be optimized for all specific use cases of the backup system.

3.2.4 Using inotify for File System Monitoring

Status
Accepted
Context
We envision Phoenix, an automatic backup system that uploads files to a configurable secondary cloud-based storage. The application requires a file system monitoring functionality to detect changes in files and directories within the directories being monitored. It needs to be able to listen to an event that is caused because of file or folder modification, creation, deletion, or change in file structure. The subsequent action of backup must be triggered upon any of the events and for this reason, the subsystem responsible for monitoring must be efficient in terms of resource usage and error handling, accurate and not give false positives, and handle large volumes of file system events.
Alternatives Considered:

- inotify
- fswatch

Decision
In this context, we propose using a **inotify** by taking note of the following:

- inotify is highly efficient in terms of resource usage and event handling. It operates asynchronously, allowing applications to monitor file system events without the need for continuous polling.
- inotify provides real-time notification of file system events, allowing applications to respond promptly to changes in the watched directory.
- inotify supports a wide range of event types, including file creations, modifications, deletions, and directory modifications.
- inotify is file system agnostic and can monitor changes on different types of file systems supported by the operating system.
- inotify can be integrated into the application seamlessly with the help of the python library pyinotify.
- In comparision, `fswatch` can be complex and time-consuming, especially when configuring it to monitor specific directories or files.
- While fswatch supports multiple operating systems like macOs and Windows other than Linux, unlike pyinotify which supports only linux, it fires multiple events for a single move operation.

Consequences

- **Efficiency**
- **Real-time Event Notification**
- **Scalability**: inotify is designed to handle large volumes of file system events efficiently, making it suitable for applications that require scalable event monitoring capabilities.

- **Performance**: Pyinotify offers better performance in terms of setup time and event handling.
- **Consistent Behavior**: Pyinotify provides consistent event handling, avoiding issues such as firing multiple events for a single move operation.
- **Low Latency**: inotify provides low-latency event notification, ensuring that applications like Phoenix can respond quickly to file system changes.

3.2.5 Multi-System Backup Strategy

Status Accepted
Context We are tasked with creating a backup system that allows users to back up files or directories periodically or in response to an event. The system should offer multiple backup options, both local and cloud-based, and should be scalable, cost-effective, secure, and easy to use. The project timeline is one week, and it should be engineered and maintained easily.
Considered Alternatives:

- OneDrive (Azure):
  - Configuration and usability moderately complex.
  - 5 GB free storage limit; larger storage requires Microsoft 365 subscription.
- Google Drive:
  - 15 GB of free storage.
  - Easy to maintain and use.
- AWS:
  - Highly scalable and flexible with Amazon S3.
  - Configuration and management complexity.
  - Variable costs based on usage; no free storage.
  - Designed for organizations rather than individual users.
- Local Storage:
  - Full control over data security and privacy.
  - Fast data access and recovery speeds.
  - Requires physical space and maintenance.
  - Vulnerable to local disasters (e.g., fire, theft) without off-site redundancy.

Decision We choose to implement a multi-system backup strategy utilizing **Local Storage, Azure, and Google Drive**. This approach ensures scalability, cost-effectiveness, security, and ease of use.
Reasons:

- **Azure**: Provides robust cloud storage with scalability and security features suitable for the project's requirements.
- **Google Drive**: Offers generous free storage and ease of use, making it an attractive option for users.
- **Local Storage**: Provides full control over data security and privacy, fast access, and recovery speeds, serving as a reliable backup option.

4 Architectural Tactics and Patterns

4.1 Architectural Tactics

4.1.1 Availability

- **Exception**

  The exception handling tactic implemented in the system ensures availability by preventing system crashes or idle states in the event of errors. Throughout the entire system, robust exception handling mechanisms are employed to gracefully manage unexpected errors or exceptional conditions.

  This tactic addresses the non-functional requirement of availability, which pertains to the system's ability to remain operational and accessible even in the face of errors or failures. By effectively handling exceptions, the system minimizes downtime and maintains its functionality, thereby ensuring continuous availability to users.

4.1.2 Performace

- **Introduce Concurrency**

  The **concurrency** tactic implemented in the system enhances performance by leveraging threads. Specifically, the file system monitoring process occurs concurrently with the rest of the system operations, allowing other functionalities to be utilized simultaneously without interruption.

  This tactic addresses the non-functional requirement of performance, particularly focusing on aspects such as responsiveness and throughput. By employing threads to manage the file system monitoring independently, the system can efficiently handle multiple tasks concurrently, thereby maximizing resource utilization and improving overall system responsiveness.

4.1.3 Security

- **Authenticate Users**

  Implementing a robust **authentication** mechanism within the architectural framework serves to validate the identity of users accessing the system. By requiring users to authenticate themselves through secure methods such as passwords, biometrics, or multi-factor authentication, this tactic enhances the security posture of the system, ensuring that only authorized individuals gain access to sensitive resources By using GPG keys for encryption and third party login from the cloud provider, we ensure that only authorized individuals access the data.

- **Maintain Data Confidentiality**

  Enforcing **data confidentiality** within the architectural design ensures that sensitive information remains inaccessible to unauthorized parties. Through encryption techniques, access controls, and secure transmission protocols, this tactic safeguards data integrity and privacy, mitigating the risk of unauthorized disclosure or exploitation. Through encryption using GPG, we ensure that data stored in the cloud is confidential, since only the encrypting user can decrypt it.

- **Limit Access**

  The **limit access** architectural tactic implemented in the system involves restricting access to specific directories within the file system monitor using Pyinotify. By explicitly specifying the directories to be watched, the system ensures that only authorized areas of the file system are monitored for changes, thereby limiting potential security risks.

  This tactic addresses the non-functional requirement of security, specifically focusing on aspects such as confidentiality, integrity, and availability. By restricting access to designated directories, the system mitigates the risk of unauthorized access or modification to sensitive files and data. It helps prevent potential security breaches, unauthorized disclosures, or tampering with critical system resources.

4.1.4 Modifiability

- **Hide Information**

  Employing the **hide information** tactic in the modifiability context involves encapsulating implementation details and minimizing dependencies to shield internal workings from external components. By abstracting complex functionalities behind well-defined interfaces and employing modular design principles, it facilitates easier maintenance, promotes system evolvability, and reduces the impact of changes. The use of strategy pattern allows us to switch between different "hows" of doing the same "what", even at runtime.

- **Use an Intermediary**

  Incorporating the **use an intermediary** tactic within the modifiability context entails introducing an intermediate layer or component between interacting modules or systems. This intermediary acts as a mediator, decoupling communication and facilitating flexible interactions. By abstracting communication protocols, managing dependencies, and providing a buffer for changes, this tactic promotes modifiability by allowing for easier adaptation, integration of new functionalities, and replacement of components without impacting the overall system architecture. The use of a broker to communicate between subsystems displays this tactic. Modification to flow of the system can be achieved by simply modifying the broker functions called by the subsystems.

4.1.5 Testability

- **Separate Interface from Implementation**

  Implementing the **Separate Interface from Implementation** tactic within the realm of testability involves decoupling the interface of a software component from its underlying implementation details. By defining clear and standardized interfaces that specify the expected behavior of the component, testers can develop test cases independently of the implementation details. This tactic enhances testability by enabling isolated testing of components through unit tests, facilitating the creation of mock objects and promoting test reusability across different implementations.

4.1.6 Usability

- **Separate UI from Rest of System**

  The **Separate UI from Rest of System** tactic, implemented using command-line interface (CLI), focuses on improving usability by decoupling the user interface (UI) from the core functionalities of the system.

4.2 Implementation Patterns

4.2.1 Strategy in Compression and Encryption Subsystem

See Figure 3
Context: `CEManager`
Strategy: `ICompressionStrategy`
Concrete strategies: `NoCompressionStrategy`, `TarCompressionStrategy`

4.2.2 Strategy and Singleton in Compression and Encryption Subsystem

See Figure 4
Context: `CEManager`
Strategies: `IEncryptionStrategy`, `IKeyStrategy`
Concrete strategies: `GPGEncryptionStrategy`, `GPGKeyStrategy`
Singletons: `KeyManager`, `GPGSingleton`

4.2.3 Strategy Pattern in Observation Subsystem

See Figure 5
Client: `CLI`
Context: `OManager`

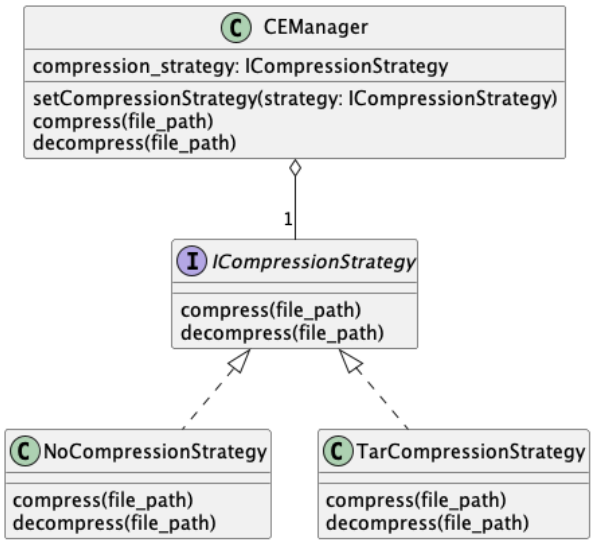

Figure 3: Strategy pattern in compression

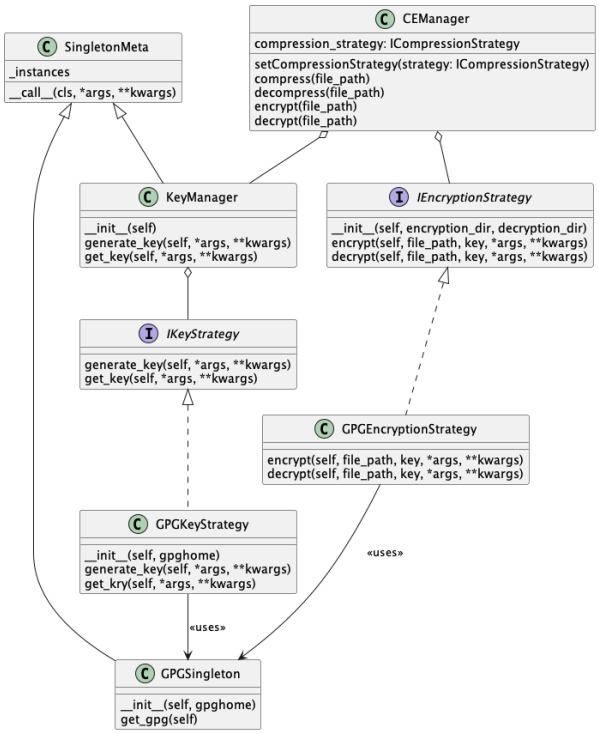

Figure 4: Strategy and singleton patterns in encryption

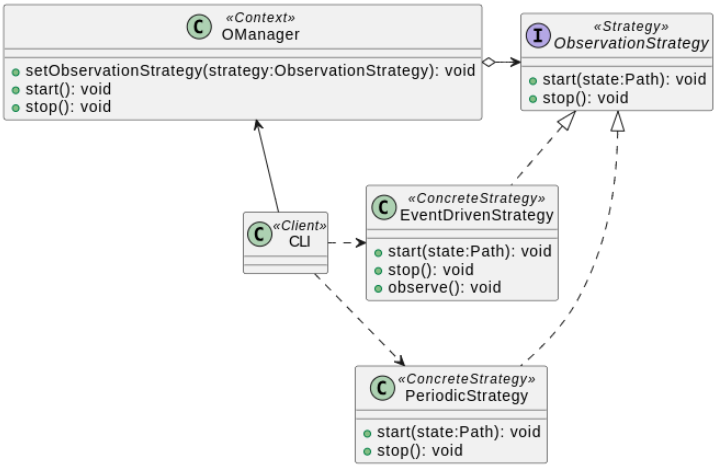

Figure 5: Strategy pattern to Triggering a backup

Strategy: `ObservationStrategy`
Concrete strategies: `EventDrivenStrategy`, `PeriodicStrategy`

4.2.4 Composite Pattern in Observation Subsystem

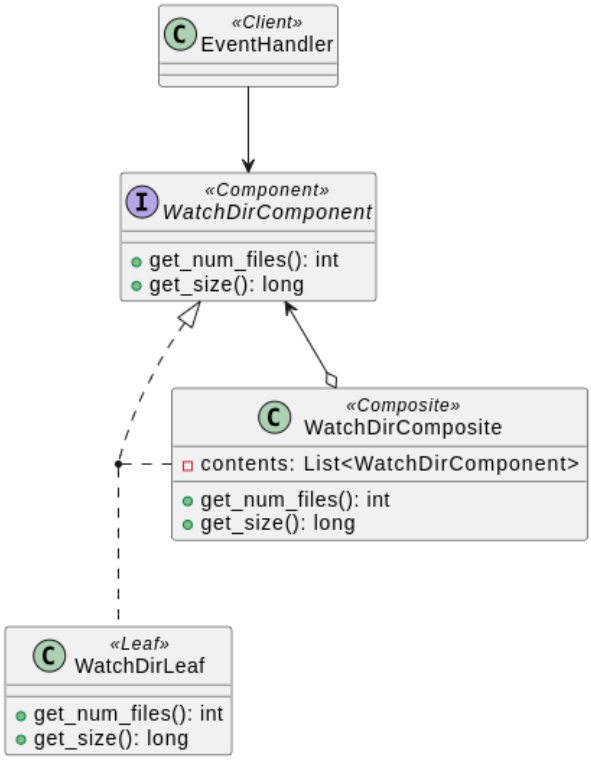

Figure 6: Composite pattern to get File System Details

See Figure 6
Client: `EventHandler`
Component: `WatchDirComponent`
Composite: `WatchDirComposite`
Leaf: `WatchDirLeaf`

4.2.5 Strategy Pattern in Uploader and Downloader Subsystem

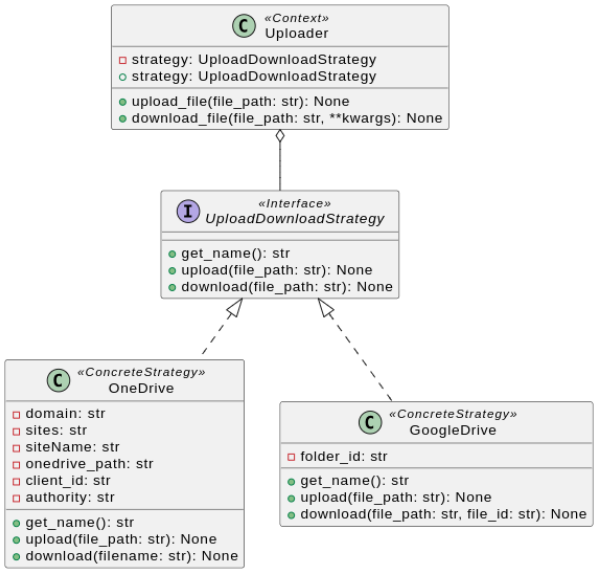

Figure 7: Strategy Pattern for Uploading and Downloading Files

See Figure 7
Context: `Uploader`
Strategy: `UploadDownloadStrategy`
Concrete strategies: `OneDrive`, `GoogleDrive`

5 Prototype Implementation and Analysis

5.1 Prototype Development

GitHub Repo Link

5.2 Architecture Analysis

We perform architecture analysis comparing two types of architectures - Monolith and Microservices. For comparison between the two, we do analysis over these non-functional requirements -

1. **Response Time:** The response time in both architectures was measured by sending request and checking the time it took for modification of file and completing the backup.

   The predictability of response times was hindered by the unpredictability of file sizes for backup and the variability in network latency within the system.
   However, despite these uncertainties, distinct differences in response times were observed between Monolithic and Microservices architectures. Microservices incurred additional overhead due to the need for network communication between various services. In contrast, Monolithic architecture, which relies on simple function calls, demonstrated notably faster performance.

2. **Extensibility:** Monolithic architecture consolidates all functionalities into a single unit, making it challenging to extend as changes require modifying the entire codebase, which introduces risk and complexity. Conversely, microservices architecture is structured as a collection of loosely coupled services, each handling a specific functionality, which inherently supports high extensibility. Developers can independently develop, deploy, and scale these services, simplifying updates and integration of new features. This design also allows for using different programming languages and technologies, enhancing adaptability and making microservices the preferable choice for systems that require frequent updates or expansions in functionality.

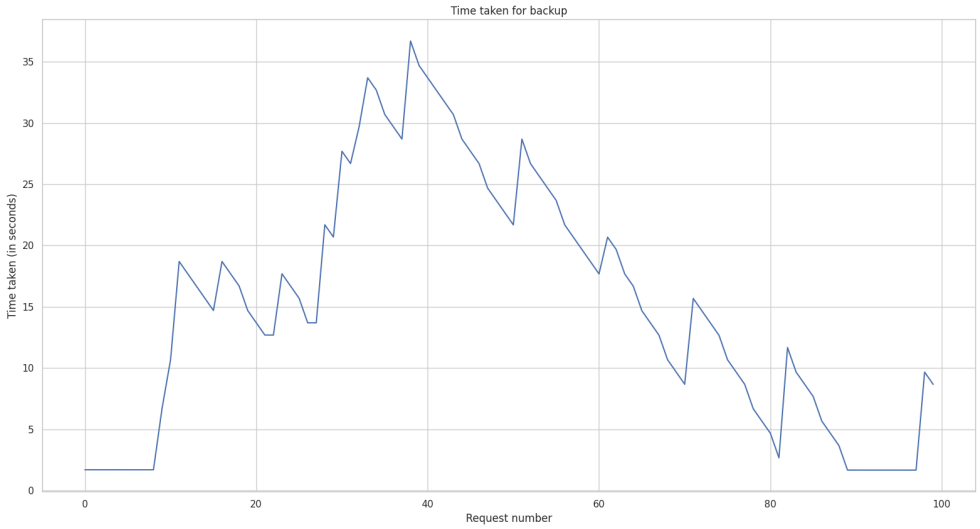

Figure 8: Response Times for Monolith Architecture

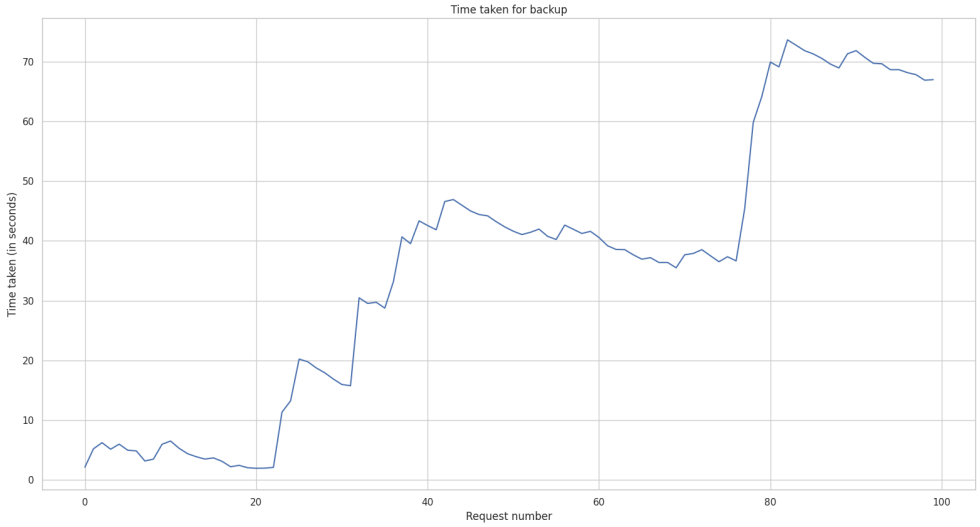

Figure 9: Response Times for Microservices Architecture

3. **Configurability:** Monolithic architecture, where all components are integrated into a single application, tends to limit configurability options. Any changes in configuration, such as adjusting encryption or compression settings, typically apply globally and affect the entire system, which can be restrictive if different parts of the application require different configurations. On the other hand, microservices architecture offers enhanced configurability due to its decentralized nature. Each microservice can be configured independently, allowing for tailored settings that suit specific needs without impacting other services. This flexibility makes it easier to adhere to varying user preferences or policies, and to adjust these settings dynamically as requirements evolve. Therefore, for systems where configurable settings are critical and likely to change, microservices architecture provides a more adaptable and suitable framework than its monolithic counterpart.

6 Contribution

- Adyansh Kakran: Observation Subsystem, CLI and Microservice Architecture
- Meghana Tedla: Observation Subsystem, CLI and Microservice Architecture, Benchmarking
- Prakhar Jain: Storage Subsystem, Benchmarking
- Rudra Dhar: Storage Subsystem
- Shrikara Arun: Compression and Encryption subsystem, CLI

Each person made requirements, overviews, tactics, and patterns for their subsystem. ADRs were made in collaboration when necessary.