## Table of Contents

* [FastAPI Server Documentation](#section-0)
* [System Overview](#section-1)
* [FastAPI Server Architecture](#section-2)
* [Design Patterns](#section-3)
* [Provider System](#section-4)
* [File Processing Workflow](#section-5)
* [Security Implementation](#section-6)
* [AWS Integration](#section-7)
* [Error Handling](#section-8)

# FastAPI Server Documentation

This documentation provides a comprehensive overview of the FastAPI server implemented in the Octo project. It covers the server architecture, workflow, design patterns, and key components.

## Table of Contents

1. [System Overview](./01-system-overview.md)
2. [FastAPI Server Architecture](./02-fastapi-architecture.md)
3. [Design Patterns](./03-design-patterns.md)
4. [Provider System](./04-provider-system.md)
5. [File Processing Workflow](./05-file-processing.md)
6. [Security Implementation](./06-security.md)
7. [AWS Integration](./07-aws-integration.md)
8. [Error Handling](./08-error-handling.md)

## Additional Resources

* [Word Document](./octo-fastapi-documentation.docx): Contains the complete documentation in a single file.
* [Images Directory](./images/): Contains all diagrams used in the documentation.

## Getting Started

To understand the overall system, start with the [System Overview](./01-system-overview.md) document. For developers implementing new providers or extending the system, the [Design Patterns](./03-design-patterns.md) and [Provider System](./04-provider-system.md) documents will be most relevant.

# System Overview

## Introduction

The Octo project is a data processing system designed to automate the retrieval, decryption, and storage of financial data from various providers like Affirm and Bloomberg. The system is built around a FastAPI server that manages asynchronous jobs to process files from these providers.

## System Architecture

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐ │ │ │ │ │ │ │ External API │ │ FastAPI │ │ AWS S3 │ │ Providers │◄────┤ Server ├────►│ Storage │ │ (Affirm, │ │ │ │ │ │ Bloomberg) │ │ │ │ │ └─────────────────┘ └────────┬────────┘ └─────────────────┘ │ │ ▼ ┌─────────────────┐ │ │ │ Data Analysis │ │ Scripts │ │ │ └─────────────────┘

The system consists of the following key components:

1. **FastAPI Server**: The core of the system, handling API requests, authentication, and job management.
2. **Provider System**: Handles connections to external data providers (Affirm, Bloomberg) via SFTP.
3. **File Processing Pipeline**: Manages the download, decryption, decompression, and upload of files.
4. **AWS S3 Integration**: Stores processed files in an organized structure.
5. **Data Analysis Scripts**: Various Python and Jupyter notebook scripts for analyzing the financial data.

## Key Workflows

### Main Data Processing Workflow

┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐ │ │ │ │ │ │ │ │ │ │ │ Connect │ │ Download │ │ Decrypt │ │ Process │ │ Upload │ │ to SFTP ├───►│ Files ├───►│ Files ├───►│ Files ├───►│ to S3 │ │ │ │ │ │ │ │ │ │ │ └──────────┘ └──────────┘ └──────────┘ └──────────┘ └──────────┘

1. The server connects to the provider's SFTP server
2. Files are downloaded from the provider
3. Files are decrypted (if encrypted with GPG)
4. Files are processed (unzipped, transformed as needed)
5. Processed files are uploaded to AWS S3 for storage and further analysis

### API Job Management Workflow

┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐ │ │ │ │ │ │ │ │ │ API │ │ Create │ │ Execute │ │ Update │ │ Request ├───►│ Job ├───►│ Job │───►│ Status │ │ │ │ │ │ Task │ │ │ └──────────┘ └──────────┘ └──────────┘ └──────────┘

1. API receives a request to process files from a specific provider
2. A job is created and tracked in the system
3. The job is executed as a background task
4. Job status is updated as processing progresses

## Technology Stack

* **Backend Framework**: FastAPI
* **Authentication**: OAuth2 with API key authentication
* **Asynchronous Processing**: Python asyncio
* **File Transfer**: Paramiko (SSH/SFTP)
* **File Encryption**: GnuPG
* **Cloud Storage**: AWS S3
* **Data Analysis**: Python, Pandas, NumPy, etc.

## Deployment

The system is designed to be deployed on an AWS EC2 instance using Docker. The SETUP.MD file provides detailed instructions for setting up the environment.

## Next Steps

For more detailed information about specific components, refer to the following documents:

* [FastAPI Server Architecture](./02-fastapi-architecture.md)
* [Design Patterns](./03-design-patterns.md)
* [Provider System](./04-provider-system.md)

# FastAPI Server Architecture

## Introduction

The FastAPI server is the central component of the Octo system, providing a modern, high-performance API for triggering and managing data processing jobs. This document details the architecture and implementation of the FastAPI server.

## Server Structure

┌─────────────────────────────────────────────────────────────┐ │ FastAPI Application │ │ │ │ ┌─────────────────┐ ┌─────────────────────────┐ │ │ │ │ │ │ │ │ │ API Endpoints │◄─────────►│ Background Task System │ │ │ │ │ │ │ │ │ └─────────────────┘ └─────────────────────────┘ │ │ │ │ │ │ │ │ │ │ ▼ ▼ │ │ ┌─────────────────┐ ┌─────────────────────────┐ │ │ │ │ │ │ │ │ │ Authentication │ │ Job Management │ │ │ │ │ │ │ │ │ └─────────────────┘ └─────────────────────────┘ │ │ │ │ │ │ │ │ ▼ │ │ ┌─────────────────────────┐ │ │ │ │ │ │ │ Main Logic Processing │ │ │ │ │ │ │ └─────────────────────────┘ │ │ │ └─────────────────────────────────────────────────────────────┘

## Key Components

### 1. API Endpoints

The FastAPI server exposes two main endpoints:

* **POST /** - Creates a new job for a specified provider
  + Requires authentication via API key
  + Takes a RootBody object containing the provider name
  + Returns a job object with a unique ID and initial status
* **GET /tasks** - Lists all current jobs and their statuses
  + Requires authentication via API key
  + Returns a dictionary of all active jobs

@app.post("/", dependencies=[Depends(api\_key\_auth)]) def test(background\_tasks: BackgroundTasks, body: RootBody): print('body', body); existing\_jobs = [job for job in jobs.values() if job.provider == body.provider] if existing\_jobs: return {"message": "A job with the same provider is already in progress"} new\_task = Job() new\_task.provider = body.provider jobs[new\_task.uid] = new\_task background\_tasks.add\_task(start\_new\_task, new\_task.uid, 100, body.provider) return new\_task @app.get("/tasks", dependencies=[Depends(api\_key\_auth)]) async def status\_handler(): return jobs

### 2. Authentication System

The server uses OAuth2 with API key authentication:

# API key storage api\_keys = [ data.get('API\_KEY') ] oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token") def api\_key\_auth(api\_key: str = Depends(oauth2\_scheme)): if api\_key not in api\_keys: raise HTTPException( status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Forbidden" )

The API key is loaded from environment variables and validated on each request using FastAPI's dependency injection system.

### 3. Job Management System

Jobs are represented using Pydantic models and stored in memory:

class Job(BaseModel): uid: UUID = Field(default\_factory=uuid4) status: str = "in\_progress" progress: int = 0 result: int = None provider: str = None jobs: Dict[UUID, Job] = {} # Dict as job storage

Each job has:

* A unique identifier (uid)
* A status field (in\_progress or complete)
* A progress percentage
* A result field (for storing any result data)
* A provider name

### 4. Background Task System

FastAPI's background tasks are used to run jobs asynchronously:

async def start\_new\_task(uid: UUID, param: int, provider: str) -> None: queue = asyncio.Queue() task = asyncio.create\_task(start\_logic(queue, provider)) while progress := await queue.get(): # monitor task progress jobs[uid].progress = progress jobs[uid].status = "complete" await asyncio.sleep(1) jobs.pop(uid)

The background task system:

1. Creates an asyncio queue for progress updates
2. Starts the main logic as an asyncio task
3. Continuously updates job progress from the queue
4. Marks the job as complete and removes it after completion

### 5. Main Processing Logic

The main processing logic is decoupled from the server and imported from the src.main\_logic module:

task = asyncio.create\_task(start\_logic(queue, provider))

This separation of concerns allows the main processing logic to evolve independently from the API server.

## Data Flow

1. Client makes API request with API key 2. Server authenticates the request 3. Server creates a new job instance 4. Server launches a background task 5. Background task runs the main processing logic 6. Main logic reports progress via queue 7. Background task updates job status 8. Client can check job status via GET /tasks endpoint

## Concurrency and Performance Considerations

The FastAPI server leverages Python's asyncio to handle concurrent requests efficiently. Key performance aspects include:

1. **Asynchronous Task Processing**:
   * Jobs run asynchronously as background tasks
   * The main API routes remain responsive even during long-running jobs
2. **One Job Per Provider**:
   * The system prevents multiple jobs for the same provider from running simultaneously
   * This prevents race conditions when accessing provider resources
3. **In-Memory Job Storage**:
   * Jobs are stored in memory for fast access
   * Jobs are automatically removed when completed to prevent memory leaks
4. **Progress Reporting**:
   * Uses an asyncio Queue for efficient progress updates
   * Progress is reported as a percentage, allowing clients to track job completion

## Error Handling

The server includes several error handling mechanisms:

1. **API Authentication Errors**:
   * Invalid API keys result in a 401 Unauthorized response
2. **Provider Validation**:
   * Non-existent providers are detected early in the process
3. **Job Duplication Prevention**:
   * Attempts to create duplicate jobs for the same provider are rejected
4. **Exception Handling in Tasks**:
   * The main logic includes exception handling to prevent unhandled exceptions from crashing the server

## Future Enhancements

Potential enhancements to the FastAPI server architecture:

1. **Persistent Job Storage**:
   * Implement a database backend for jobs to survive server restarts
2. **Job Cancellation**:
   * Add an endpoint to cancel running jobs
3. **Scheduled Jobs**:
   * Add support for scheduled/recurring jobs
4. **Detailed Job Logs**:
   * Enhance job objects to include detailed logs and error information

# Design Patterns

## Introduction

The Octo FastAPI server implementation uses several design patterns to achieve a clean, maintainable architecture. This document outlines the key design patterns used in the system and explains how they contribute to the overall architecture.

## Factory Pattern

### Implementation

The Factory Pattern is used to create provider-specific handlers through the ProviderFactory class:

class ProviderFactory(): def \_\_init\_\_(self, provider): self.provider = provider def get\_provider(self, data) -> BaseProvider | None: if self.provider == "affirm": from src.core.providers.affirm import AffirmProvider return AffirmProvider(data) elif self.provider == "bloomberg": from src.core.providers.bloomberg import BloombergProvider return BloombergProvider(data) else: return None

### Usage

The factory is used in the main logic to create the appropriate provider instance:

providerObject = ProviderFactory(provider\_str) provider = providerObject.get\_provider(data)

### Benefits

* **Encapsulation**: Hides the complexity of provider creation
* **Extensibility**: New providers can be added by extending the factory without modifying client code
* **Conditional Loading**: Only imports the necessary provider modules when needed

## Strategy Pattern

### Implementation

The Strategy Pattern is implemented through the BaseProvider abstract class and its concrete implementations:

class BaseProvider: def \_\_init\_\_(self, data: dict[str, str | None], provider\_name: str): self.data = data self.provider\_name = provider\_name self.sftp\_client = None self.ssh\_client = None def get\_bucket\_name(): pass def connect\_ssh(): pass # other abstract methods...

Concrete implementations for different providers:

class AffirmProvider(BaseProvider): def \_\_init\_\_(self, data): super().\_\_init\_\_(data, 'affirm') def get\_bucket\_name(self): return self.data.get('S3\_BUCKET\_AFFIRM') # other overridden methods...

### Usage

The client code (main logic) interacts with the abstract BaseProvider interface, unaware of the specific provider implementation:

await provider.get\_save\_files(bucket\_name) ssh\_client = await provider.connect\_ssh() pending\_files = await provider.get\_pending\_files(saved\_files\_arr) # ...

### Benefits

* **Interchangeability**: Different provider implementations can be used interchangeably
* **Separation of Concerns**: Each provider encapsulates its specific behavior
* **Extensibility**: New providers can be added without changing the client code

## Template Method Pattern

### Implementation

The Template Method Pattern is implemented in the BaseProvider class, which defines a skeleton for the file processing algorithm while allowing subclasses to override specific steps:

class BaseProvider: # ... other methods ... async def decrypt\_file\_raw(self, file\_path: str, passphrase: str): file\_decrypted = file\_path if '.gpg' in file\_path: file\_decrypted = await decrypt\_file(file\_path, passphrase) file\_name = file\_decrypted return file\_name async def unzip\_file(self, file\_decrypted: str): file\_name = file\_decrypted if '.gz' in file\_decrypted or '.tar.gz' in file\_decrypted: file\_name = remove\_final\_extension(file\_name) file\_extension = '.tar.gz' if file\_name.endswith('.tar.gz') else '.gz' await unzip\_file(file\_decrypted, file\_name, file\_extension) move\_file(file\_name, 'output/'+file\_name) return file\_name

### Usage

Provider-specific classes can override parts of this process while maintaining the overall algorithm structure.

### Benefits

* **Code Reuse**: Common processing steps are defined once in the base class
* **Consistency**: Ensures a consistent approach across different providers
* **Flexibility**: Allows customization of specific steps when needed

## Dependency Injection

### Implementation

Dependency Injection is used throughout the codebase to provide configuration data, queues, and other dependencies to components:

# Injecting configuration data def \_\_init\_\_(self, data: dict[str, str | None], provider\_name: str): self.data = data self.provider\_name = provider\_name # Injecting a queue for progress updates async def start\_logic(queue: asyncio.Queue, provider\_str: str): # ... await queue.put(float("{:.2f}".format(percentage)))

### Usage

Dependencies are passed explicitly to functions and classes rather than being created or retrieved internally.

### Benefits

* **Testability**: Components can be tested in isolation with mock dependencies
* **Flexibility**: Dependencies can be swapped without changing component code
* **Decoupling**: Reduces tight coupling between components

## Observer Pattern (via Queue)

### Implementation

The Observer Pattern is implemented through the use of an asyncio Queue to report progress from the main logic back to the background task:

async def start\_new\_task(uid: UUID, param: int, provider: str) -> None: queue = asyncio.Queue() task = asyncio.create\_task(start\_logic(queue, provider)) while progress := await queue.get(): # monitor task progress jobs[uid].progress = progress jobs[uid].status = "complete" await asyncio.sleep(1) jobs.pop(uid)

### Usage

The main logic function acts as the subject, publishing progress updates to the queue, while the background task acts as the observer, updating the job status based on these updates.

### Benefits

* **Loose Coupling**: The main logic doesn't need to know about the job tracking system
* **Asynchronous Updates**: Progress can be reported asynchronously without blocking the main processing
* **Separation of Concerns**: Processing logic is separated from progress tracking

## Command Pattern (via Background Tasks)

### Implementation

The Command Pattern is evident in the way background tasks are created and executed:

background\_tasks.add\_task(start\_new\_task, new\_task.uid, 100, body.provider)

### Usage

The API endpoint creates a command (background task) that encapsulates the request and its parameters, which is then executed independently.

### Benefits

* **Asynchronous Execution**: Commands run without blocking the API response
* **Queueing**: FastAPI's background task system handles queuing and execution
* **Decoupling**: Command execution is decoupled from API request handling

## Proxy Pattern (SSH/SFTP)

### Implementation

The Proxy Pattern is used to provide a common interface for accessing remote files over SSH/SFTP:

@sync\_to\_async def connect\_ssh\_raw( self, server\_env: str, user\_name: str, hostname: str, port: int, private\_key\_path: str | None = None, password: str | None = None, ): ssh\_client = paramiko.SSHClient() ssh\_client.load\_system\_host\_keys() # ... setup and connect ... return ssh\_client

### Usage

The SFTP/SSH connection acts as a proxy for files on remote systems, providing a local interface to remote resources.

### Benefits

* **Access Control**: Centralized handling of authentication and access
* **Abstraction**: Simplifies interactions with remote file systems
* **Resource Management**: Centralizes connection management and cleanup

## Adapter Pattern

### Implementation

The Adapter Pattern is used to adapt various file formats (encrypted, compressed) to a common interface:

async def decrypt\_file\_raw(self, file\_path: str, passphrase: str): file\_decrypted = file\_path if '.gpg' in file\_path: file\_decrypted = await decrypt\_file(file\_path, passphrase) file\_name = file\_decrypted return file\_name async def unzip\_file(self, file\_decrypted: str): # ... handle various compression formats ...

### Usage

These adapter methods convert files from their original format to a format that can be processed by the system, regardless of how they were originally stored.

### Benefits

* **Format Independence**: Processing logic can work with files regardless of their original format
* **Simplification**: Complex format handling is encapsulated in adapter methods
* **Extensibility**: New file formats can be supported by adding new adapter methods

## Design Patterns Interaction

The design patterns in the system don't exist in isolation but work together to create a cohesive architecture:

┌──────────────────┐ ┌──────────────────┐ │ │ │ │ │ Factory Pattern │───creates────►│ Strategy Pattern │ │ │ │ │ └──────────────────┘ └──────────────────┘ │ │ implements ▼ ┌──────────────────┐ ┌──────────────────┐ │ │ │ │ │ Command Pattern │◄───triggers───│ Template Method │ │ │ │ │ └──────────────────┘ └──────────────────┘ │ │ │ │ ▼ ▼ ┌──────────────────┐ ┌──────────────────┐ │ │ │ │ │ Observer Pattern │◄───updates────│ Adapter Pattern │ │ │ │ │ └──────────────────┘ └──────────────────┘ │ │ ▼ ┌──────────────────┐ │ │ │ Proxy Pattern │ │ │ └──────────────────┘

## Conclusion

The Octo project demonstrates the effective use of multiple design patterns to create a flexible, maintainable architecture. These patterns work together to:

1. **Decouple Components**: Each component has a single responsibility and minimal knowledge of other components
2. **Facilitate Extension**: New providers and file types can be added with minimal changes to existing code
3. **Promote Reuse**: Common functionality is encapsulated and reused across the system
4. **Improve Testability**: Components can be tested in isolation with mock dependencies

By understanding these design patterns and their interactions, developers can maintain and extend the system more effectively, following the established architectural principles.

# Provider System

## Introduction

The Provider System is a core component of the Octo application, responsible for handling the interaction with different data providers such as Affirm and Bloomberg. This document details the architecture of the Provider System, how it works, and how to extend it with new providers.

## System Architecture

┌───────────────────────────────────────────────────────────────────┐ │ Provider System │ │ │ │ ┌───────────────────┐ ┌───────────────────────────────┐ │ │ │ │ │ │ │ │ │ ProviderFactory │───────►│ BaseProvider │ │ │ │ │ creates│ (Abstract Provider Class) │ │ │ └───────────────────┘ └─────────────────┬─────────────┘ │ │ │ │ │ │ extends │ │ │ │ │ ┌─────────────────────────────────────────┐ │ │ │ │ │ │ │ │ │ │ ┌─────────────────▼─────┐ ┌─────────────────────▼─┐ │ │ │ │ │ │ │ │ │ AffirmProvider │ │ BloombergProvider │ │ │ │ │ │ │ │ │ └───────────────────────┘ └───────────────────────┘ │ │ │ │ Concrete Providers │ │ │ └───────────────────────────────────────────────────────────────────┘

## Component Details

### Provider Factory

The ProviderFactory class is responsible for creating the appropriate provider instance based on the provider name:

class ProviderFactory(): def \_\_init\_\_(self, provider): self.provider = provider def get\_provider(self, data) -> BaseProvider | None: if self.provider == "affirm": from src.core.providers.affirm import AffirmProvider return AffirmProvider(data) elif self.provider == "bloomberg": from src.core.providers.bloomberg import BloombergProvider return BloombergProvider(data) else: return None

**Key Features:**

* Lazy loading of provider modules (only imports what's needed)
* Single responsibility of creating provider instances
* Extensible design for adding new providers

### Base Provider

The BaseProvider class is an abstract base class that defines the interface for all provider implementations:

class BaseProvider: def \_\_init\_\_(self, data: dict[str, str | None], provider\_name: str): self.data = data self.provider\_name = provider\_name self.sftp\_client = None self.ssh\_client = None def get\_bucket\_name(): pass def connect\_ssh(): pass async def sftp\_download(self, file\_path: str): pass def sft\_list\_inbox(self): pass async def get\_pending\_files(self, saved\_files\_arr:list[str]): pass def get\_sub\_folder(self, file\_name: str): pass def upload\_file(self, file\_name: str): pass def decrypt\_file(): pass # Other methods...

**Key Features:**

* Defines the contract that all providers must implement
* Provides common utility methods for file handling
* Manages shared resources like SSH connections

### Concrete Providers

Concrete provider classes implement the specific logic for interacting with each provider:

**AffirmProvider Example:**

class AffirmProvider(BaseProvider): def \_\_init\_\_(self, data): super().\_\_init\_\_(data, 'affirm') def get\_bucket\_name(self): return self.data.get('S3\_BUCKET\_AFFIRM') # Other provider-specific implementations...

**Key Features:**

* Provider-specific configuration (bucket names, connection details)
* Custom file handling logic if needed
* Proper implementation of all required interface methods

## Provider Workflow

Each provider follows the same general workflow, with provider-specific implementations for each step:

┌──────────────┐ │ │ │ Connect to │ │ Provider │ │ │ └──────┬───────┘ │ ▼ ┌──────────────┐ │ │ │ List Files │ │ to Process │ │ │ └──────┬───────┘ │ ▼ ┌──────────────┐ │ │ │ Download │ │ Files │ │ │ └──────┬───────┘ │ ▼ ┌──────────────┐ │ │ │ Decrypt │ │ Files │ │ │ └──────┬───────┘ │ ▼ ┌──────────────┐ │ │ │ Process │ │ Files │ │ │ └──────┬───────┘ │ ▼ ┌──────────────┐ │ │ │ Upload to S3 │ │ │ └──────────────┘

### Connection to Provider

Each provider implements its own connection method:

async def connect\_ssh(self): hostname = self.data.get('SFTP\_HOSTNAME\_PROVIDER') username = self.data.get('SFTP\_USERNAME\_PROVIDER') port = int(self.data.get('SFTP\_PORT\_PROVIDER')) server\_env = self.data.get('SERVER\_ENV') private\_key\_path = self.data.get('PRIVATE\_KEY\_PATH\_PROVIDER') self.ssh\_client = await self.connect\_ssh\_raw(server\_env, username, hostname, port, private\_key\_path) return self.ssh\_client

### File Listing

Providers implement logic to list files that need to be processed:

async def get\_pending\_files(self, saved\_files\_arr: list[str]): sftp\_client = await self.sft\_list\_inbox() remote\_files = sftp\_client.listdir("inbox/") return items\_not\_contained\_in\_list(remote\_files, saved\_files\_arr)

### File Download

Each provider downloads files from its source:

async def sftp\_download(self, file\_path: str): sftp\_client = paramiko.SFTPClient.from\_transport(self.ssh\_client.get\_transport()) print(f"Downloading {file\_path}") sftp\_client.get(f"inbox/{file\_path}", file\_path) sftp\_client.close()

### File Decryption

Providers handle decryption if files are encrypted:

async def decrypt\_file(self, file\_path: str): passphrase = self.data.get('GPG\_PASSPHRASE\_PROVIDER') file\_name = await self.decrypt\_file\_raw(file\_path, passphrase) return file\_name

### File Upload

After processing, files are uploaded to S3:

async def upload\_file(self, file\_name: str): bucket\_name = self.get\_bucket\_name() await self.upload\_file\_raw(file\_name, bucket\_name)

## Extending the Provider System

### Adding a New Provider

To add a new provider to the system:

1. **Create a new provider class** that extends BaseProvider:

# src/core/providers/new\_provider.py from src.core.providers.base import BaseProvider class NewProvider(BaseProvider): def \_\_init\_\_(self, data): super().\_\_init\_\_(data, 'new\_provider') def get\_bucket\_name(self): return self.data.get('S3\_BUCKET\_NEW\_PROVIDER') # Implement all required methods...

1. **Update the factory** to create instances of the new provider:

def get\_provider(self, data) -> BaseProvider | None: if self.provider == "affirm": from src.core.providers.affirm import AffirmProvider return AffirmProvider(data) elif self.provider == "bloomberg": from src.core.providers.bloomberg import BloombergProvider return BloombergProvider(data) elif self.provider == "new\_provider": from src.core.providers.new\_provider import NewProvider return NewProvider(data) else: return None

1. **Add environment variables** for the new provider in .env or environment configuration.
2. **Test the new provider** by sending a request with the new provider name.

### Provider Requirements

When implementing a new provider, ensure it:

1. **Extends BaseProvider**: Inherits and implements all required methods
2. **Handles Authentication**: Implements correct authentication for the provider
3. **Manages Resources**: Properly acquires and releases resources (connections, files)
4. **Reports Progress**: Uses the queue to report progress during processing
5. **Handles Errors**: Includes appropriate error handling and reporting

## Configuration

Providers are configured through environment variables loaded via the get\_env\_data() function:

data = get\_env\_data() providerObject = ProviderFactory(provider\_str) provider = providerObject.get\_provider(data)

Common configuration parameters include:

* **S3\_BUCKET\_PROVIDER**: S3 bucket name for storing files
* **SFTP\_HOSTNAME\_PROVIDER**: SFTP server hostname
* **SFTP\_USERNAME\_PROVIDER**: SFTP username
* **SFTP\_PORT\_PROVIDER**: SFTP port number
* **SERVER\_ENV**: Environment (dev/prod)
* **PRIVATE\_KEY\_PATH\_PROVIDER**: Path to SSH private key
* **GPG\_PASSPHRASE\_PROVIDER**: Passphrase for GPG decryption

## Error Handling

Providers should implement robust error handling:

try: # Provider operation except Exception as e: print(f"an error occurred while processing file {file\_path}: {e}") # Clean up resources, report error, continue if possible

The main logic also includes error handling to prevent provider errors from crashing the entire process.

## Best Practices

When working with the Provider System:

1. **Follow the Interface**: Implement all methods defined in BaseProvider
2. **Resource Management**: Always close connections and clean up files
3. **Error Handling**: Handle and report errors gracefully
4. **Progress Reporting**: Update the progress queue regularly
5. **Idempotent Operations**: Ensure operations can be safely retried
6. **Minimal Dependencies**: Keep provider implementations focused on their specific tasks

## Testing

Providers should be tested to ensure they:

1. Connect to their data sources correctly
2. List and filter files properly
3. Download, decrypt, and process files as expected
4. Upload files to S3 with the correct structure
5. Handle errors gracefully

Use mocking to test providers without actual external connections during development.

# File Processing Workflow

## Introduction

The file processing workflow is a central component of the Octo system, responsible for downloading, decrypting, processing, and storing files from various providers. This document details the end-to-end file processing workflow and each step involved.

## Overall Workflow

┌───────────────┐ ┌───────────────┐ ┌───────────────┐ ┌───────────────┐ ┌───────────────┐ │ │ │ │ │ │ │ │ │ │ │ Determine │ │ Download │ │ Decrypt │ │ Process │ │ Upload │ │ Files to ├───►│ Files from ├───►│ Files ├───►│ Files ├───►│ to S3 │ │ Process │ │ Provider │ │ (if needed) │ │ (if needed) │ │ Storage │ │ │ │ │ │ │ │ │ │ │ └───────────────┘ └───────────────┘ └───────────────┘ └───────────────┘ └───────────────┘

The workflow begins when a job is triggered via the API, which results in a call to the start\_logic function with a provider name. The function then orchestrates the entire file processing workflow.

## Main Logic Implementation

The main orchestration logic is implemented in the start\_logic function:

async def start\_logic(queue: asyncio.Queue, provider\_str: str): try: start\_time = time.time() data = get\_env\_data() print('start logic') providerObject = ProviderFactory(provider\_str) provider = providerObject.get\_provider(data) if provider is None: print(f"Provider {provider\_str} not found") await queue.put(None) return bucket\_name = provider.get\_bucket\_name() print('bucket\_name', bucket\_name) saved\_files = await provider.get\_save\_files(bucket\_name) saved\_files\_arr = [] for file in saved\_files: file\_array = file.split('/') saved\_files\_arr.append(file\_array[len(file\_array)-1]) ssh\_client = await provider.connect\_ssh() pending\_files = await provider.get\_pending\_files(saved\_files\_arr) count = 0 total\_pending = len(pending\_files) files\_with\_error = [] for file\_path in pending\_files: try: if file\_path.endswith('.sig'): continue start\_time\_file = time.time() await provider.sftp\_download(file\_path) file\_name = await provider.decrypt\_file(file\_path) file\_decrypted = file\_name file\_name = await provider.unzip\_file(file\_name) await provider.upload\_file(file\_name) delete\_file(file\_decrypted) delete\_file(file\_path) end\_time\_file = time.time() elapsed\_time\_file = end\_time\_file - start\_time\_file delete\_file('output/'+file\_name) count += 1 percentage = (count/total\_pending) \* 100 await queue.put(float("{:.2f}".format(percentage))) print(f"{bcolors.OKGREEN}processed file (#{count}) {file\_name} in {"{:10.2f}".format(elapsed\_time\_file)} seconds {bcolors.ENDC}") print(f"{bcolors.UNDERLINE}--------------------------------------------{bcolors.ENDC}") except Exception as e: count += 1 delete\_file(file\_path) percentage = (count/total\_pending) \* 100 await queue.put(float("{:.2f}".format(percentage))) files\_with\_error.append(file\_path) print(f"an error occurred while processing file {file\_path}: {e}") print(f"{bcolors.UNDERLINE}--------------------------------------------{bcolors.ENDC}") continue await close\_connection(ssh\_client) end\_time = time.time() elapsed\_time = end\_time - start\_time await queue.put(None) if files\_with\_error: print(f"{bcolors.FAIL}Files with errors: {files\_with\_error}{bcolors.ENDC}") else: print(f"{bcolors.OKGREEN}No files with errors{bcolors.ENDC}") print(f"{bcolors.WARNING}processed finished in {"{:10.2f}".format(elapsed\_time)} seconds {bcolors.ENDC}") except Exception as e: print(f"an error occurred: {e}") await queue.put(None) return

## Detailed Steps

### 1. Determine Files to Process

The first step is to determine which files need to be processed:

saved\_files = await provider.get\_save\_files(bucket\_name) saved\_files\_arr = [] for file in saved\_files: file\_array = file.split('/') saved\_files\_arr.append(file\_array[len(file\_array)-1]) ssh\_client = await provider.connect\_ssh() pending\_files = await provider.get\_pending\_files(saved\_files\_arr)

This involves:

1. Retrieving a list of files already processed and stored in S3
2. Connecting to the provider's SFTP server
3. Comparing the files on the SFTP server with those already processed to identify new files

### 2. Download Files

For each file that needs processing, the system downloads it from the provider:

await provider.sftp\_download(file\_path)

The download logic is provider-specific, but typically involves:

1. Establishing an SFTP connection using the provider's credentials
2. Retrieving the file from the provider's server
3. Storing it locally for further processing

### 3. Decrypt Files

If the files are encrypted (e.g., with GPG), they are decrypted:

file\_name = await provider.decrypt\_file(file\_path) file\_decrypted = file\_name

The decryption process:

1. Uses the provider-specific GPG passphrase
2. Decrypts the file using the decrypt\_file function
3. Returns the path to the decrypted file

Implementation of the decrypt\_file function in the GPG module:

async def decrypt\_file(input\_file\_path: str, passphrase: str) -> str: output\_file\_path = remove\_final\_extension(input\_file\_path) gpg = gnupg.GPG() with open(input\_file\_path, 'rb') as f: status = gpg.decrypt\_file(f, passphrase=passphrase, output=output\_file\_path) if status.ok: return output\_file\_path else: raise Exception(f"Decryption failed: {status.stderr}")

### 4. Process Files

Files may need additional processing, such as decompression:

file\_name = await provider.unzip\_file(file\_name)

The unzip\_file method in BaseProvider:

async def unzip\_file(self, file\_decrypted: str): file\_name = file\_decrypted if '.gz' in file\_decrypted or '.tar.gz' in file\_decrypted: file\_name = remove\_final\_extension(file\_name) file\_extension = '.tar.gz' if file\_name.endswith('.tar.gz') else '.gz' await unzip\_file(file\_decrypted, file\_name, file\_extension) move\_file(file\_name, 'output/'+file\_name) return file\_name

This step:

1. Identifies the compression format (if any)
2. Decompresses the file using the appropriate method
3. Moves the processed file to an output directory

### 5. Upload to S3

The processed file is uploaded to AWS S3 for storage:

await provider.upload\_file(file\_name)

This step:

1. Determines the appropriate S3 bucket and folder structure
2. Uploads the file to S3
3. Organizes files in S3 based on their type and provider

The upload\_file\_raw method in BaseProvider:

async def upload\_file\_raw(self, file\_name: str, bucket\_name: str): sub\_folder = "other" if ".csv" in file\_name: sub\_folder = "csv" elif ".pdf" in file\_name: sub\_folder = "pdf" elif ".xml" in file\_name: sub\_folder = "xml" sub\_folder = sub\_folder + '/' + get\_folder\_name(file\_name, provider=self.provider\_name) await s3\_upload\_file('output/'+file\_name, bucket\_name, 'data/'+sub\_folder+'/'+file\_name)

### 6. Cleanup

After successful processing, temporary files are cleaned up:

delete\_file(file\_decrypted) delete\_file(file\_path) delete\_file('output/'+file\_name)

Finally, the SSH connection is closed:

await close\_connection(ssh\_client)

## Progress Reporting

Throughout the file processing workflow, progress is reported via the asyncio queue:

percentage = (count/total\_pending) \* 100 await queue.put(float("{:.2f}".format(percentage)))

This allows the API to track and report job progress to clients.

## Error Handling

The workflow includes comprehensive error handling:

1. **File-Level Error Handling**:try: # Process file except Exception as e: count += 1 delete\_file(file\_path) percentage = (count/total\_pending) \* 100 await queue.put(float("{:.2f}".format(percentage))) files\_with\_error.append(file\_path) print(f"an error occurred while processing file {file\_path}: {e}") continue
2. **Workflow-Level Error Handling**:try: # Main workflow logic except Exception as e: print(f"an error occurred: {e}") await queue.put(None) return

This ensures that:

1. Errors processing individual files don't stop the entire job
2. Resources are properly cleaned up in case of errors
3. Error information is logged for troubleshooting
4. The API is notified of job completion even in error cases

## Performance Considerations

The file processing workflow is designed with several performance considerations:

1. **Asynchronous Processing**:
   * Uses asyncio for non-blocking I/O operations
   * Allows multiple files to be processed efficiently
2. **Resource Management**:
   * Files are deleted after processing to free up disk space
   * SSH connections are properly closed after use
3. **Progress Tracking**:
   * Regular progress updates allow for monitoring of long-running jobs
   * Elapsed time tracking helps identify bottlenecks
4. **Error Resilience**:
   * Failed files don't stop the entire job
   * Files with errors are tracked separately for later investigation

## Extension Points

The file processing workflow can be extended in several ways:

1. **New File Formats**:
   * Implement new decryption or decompression methods
   * Add handling for different file extensions
2. **Additional Processing Steps**:
   * Add data validation or transformation steps
   * Implement additional file processing logic
3. **Enhanced Reporting**:
   * Add more detailed progress reporting
   * Implement file-specific metrics

## Conclusion

The file processing workflow is a central component of the Octo system, providing a robust, extensible mechanism for handling files from different providers. By combining provider-specific logic with common file processing steps, the system achieves both flexibility and code reuse.

Developers working on the system should understand this workflow to effectively maintain and extend the file processing capabilities.

# Security Implementation

## Introduction

Security is a critical aspect of the Octo system, which handles potentially sensitive financial data from various providers. This document outlines the security measures implemented throughout the system to protect data and prevent unauthorized access.

## Authentication and Authorization

### API Authentication

The FastAPI server implements OAuth2 with API key authentication:

# API key storage api\_keys = [ data.get('API\_KEY') ] oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token") def api\_key\_auth(api\_key: str = Depends(oauth2\_scheme)): if api\_key not in api\_keys: raise HTTPException( status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Forbidden" )

API keys are loaded from environment variables, ensuring they are not hardcoded in the source code. All API endpoints are protected by this authentication mechanism:

@app.post("/", dependencies=[Depends(api\_key\_auth)]) def test(background\_tasks: BackgroundTasks, body: RootBody): # Handler implementation

### Provider Authentication

For provider authentication, the system uses SSH key-based authentication or username/password authentication depending on the provider's requirements:

@sync\_to\_async def connect\_ssh\_raw( self, server\_env: str, user\_name: str, hostname: str, port: int, private\_key\_path: str | None = None, password: str | None = None, ): ssh\_client = paramiko.SSHClient() ssh\_client.load\_system\_host\_keys() myfile = None if private\_key\_path is not None: myfile = paramiko.RSAKey.from\_private\_key\_file(private\_key\_path) print('server\_env', server\_env) if server\_env == 'dev': proxy = paramiko.proxy.ProxyCommand('/usr/bin/nc -x 127.0.0.1:9090 %s %d' % (hostname, port)) ssh\_client.connect(hostname=hostname, username=user\_name, pkey=myfile, password=password, sock=proxy) else : ssh\_client.connect(hostname=hostname, username=user\_name, pkey=myfile, password=password) return ssh\_client

## Data Protection

### Encrypted Data Handling

The system can handle encrypted files from providers using GPG:

async def decrypt\_file(input\_file\_path: str, passphrase: str) -> str: output\_file\_path = remove\_final\_extension(input\_file\_path) gpg = gnupg.GPG() with open(input\_file\_path, 'rb') as f: status = gpg.decrypt\_file(f, passphrase=passphrase, output=output\_file\_path) if status.ok: return output\_file\_path else: raise Exception(f"Decryption failed: {status.stderr}")

Passphrases for GPG decryption are stored as environment variables, not in the code.

### Secure File Transfer

All file transfers between providers and the system use secure protocols:

1. **SFTP over SSH** for downloading files from providers
2. **HTTPS** for uploading files to AWS S3

# SFTP download async def sftp\_download(self, file\_path: str): sftp\_client = paramiko.SFTPClient.from\_transport(self.ssh\_client.get\_transport()) print(f"Downloading {file\_path}") sftp\_client.get(f"inbox/{file\_path}", file\_path) sftp\_client.close() # S3 upload with HTTPS async def s3\_upload\_file(file\_path: str, bucket: str, object\_name: str): s3\_client = boto3.client('s3') try: s3\_client.upload\_file(file\_path, bucket, object\_name) return True except Exception as e: print(f"Error uploading to S3: {e}") return False

### Resource Cleanup

To prevent sensitive data from persisting on disk, the system cleans up temporary files after processing:

delete\_file(file\_decrypted) delete\_file(file\_path) delete\_file('output/'+file\_name)

The delete\_file function securely removes files:

def delete\_file(file\_path: str): try: if os.path.exists(file\_path): os.remove(file\_path) except Exception as e: print(f"Error deleting file {file\_path}: {e}")

## Environment Security

### Environment Variables

Sensitive information such as API keys, passphrases, and connection details are stored as environment variables:

def get\_env\_data(): return { 'API\_KEY': os.environ.get('API\_KEY'), 'S3\_BUCKET\_AFFIRM': os.environ.get('S3\_BUCKET\_AFFIRM'), 'SFTP\_HOSTNAME\_AFFIRM': os.environ.get('SFTP\_HOSTNAME\_AFFIRM'), 'SFTP\_USERNAME\_AFFIRM': os.environ.get('SFTP\_USERNAME\_AFFIRM'), 'SFTP\_PORT\_AFFIRM': os.environ.get('SFTP\_PORT\_AFFIRM'), 'PRIVATE\_KEY\_PATH\_AFFIRM': os.environ.get('PRIVATE\_KEY\_PATH\_AFFIRM'), 'GPG\_PASSPHRASE\_AFFIRM': os.environ.get('GPG\_PASSPHRASE\_AFFIRM'), # ... other environment variables }

### Secure Development Practices

The system's security is also enforced through development practices:

1. **No Hardcoded Secrets**: All sensitive data is stored in environment variables
2. **Dependency Management**: Regular updates to dependencies to address security vulnerabilities
3. **Environment Isolation**: Different configurations for development and production environments

## AWS Security Configuration

### S3 Bucket Security

The system interacts with AWS S3 buckets, which should be configured with appropriate security settings:

1. **Access Control**: S3 buckets should have appropriate bucket policies and IAM roles
2. **Encryption**: Server-side encryption should be enabled for S3 buckets
3. **Access Logging**: S3 access logging should be enabled to track access to files

### Access Management

AWS access is managed through IAM roles and policies, with the principle of least privilege:

# S3 client creation with AWS credentials from environment variables def s3\_client(): return boto3.client( 's3', aws\_access\_key\_id=os.environ.get('AWS\_ACCESS\_KEY\_ID'), aws\_secret\_access\_key=os.environ.get('AWS\_SECRET\_ACCESS\_KEY') )

## Security Considerations for Deployment

### Docker Deployment

When deploying the system using Docker (as outlined in SETUP.MD), several security considerations should be addressed:

1. **Secret Management**: Use environment variables or a secure secret management system
2. **Network Security**: Configure appropriate network access controls
3. **Container Security**: Use up-to-date Docker images and security best practices

### EC2 Instance Security

For EC2 deployment:

1. **Security Groups**: Configure restrictive security groups
2. **SSH Access**: Limit SSH access to authorized users and IP addresses
3. **Updates**: Keep the system updated with security patches

## Error Handling and Logging

Proper error handling is essential for security to prevent information leakage and ensure system resilience:

try: # Operation except Exception as e: print(f"an error occurred: {e}") # Handle error without exposing sensitive information

Logs should be carefully managed to:

1. Avoid logging sensitive information (such as API keys or passwords)
2. Provide enough detail for troubleshooting
3. Be stored securely

## Security Recommendations

To enhance the security of the system, consider implementing:

1. **Enhanced Authentication**:
   * Implement token expiration and rotation
   * Add support for OAuth2 with multiple authentication providers
2. **Improved File Security**:
   * Implement file integrity checks (checksums)
   * Add support for client-side encryption of S3 uploads
3. **Monitoring and Alerts**:
   * Add security event monitoring
   * Implement alerts for suspicious activities
4. **Penetration Testing**:
   * Regularly test the system for security vulnerabilities
   * Address identified issues promptly
5. **Audit Logging**:
   * Implement comprehensive audit logging
   * Store logs securely for compliance and forensic purposes

## Conclusion

The Octo system implements various security measures to protect data and ensure secure operations. By using secure protocols, proper authentication, and careful handling of sensitive information, the system maintains a strong security posture.

Developers should continue to prioritize security when maintaining and extending the system, following the established security practices and implementing recommended enhancements as appropriate.

# AWS Integration

## Introduction

The Octo system integrates with various AWS services to facilitate secure storage and deployment of the application. This document details how the system interacts with AWS, focusing primarily on S3 storage for processed files and EC2 for deployment.

## S3 Storage Integration

### Overview

Amazon S3 (Simple Storage Service) is used as the primary storage solution for files processed by the Octo system. Files downloaded from various providers are processed and then uploaded to S3 for long-term storage and further analysis.

┌────────────────┐ ┌────────────────┐ ┌────────────────┐ │ │ │ │ │ │ │ Provider │ ───► │ Octo System │ ───► │ AWS S3 │ │ Files │ │ Processing │ │ Storage │ │ │ │ │ │ │ └────────────────┘ └────────────────┘ └────────────────┘

### S3 Client Configuration

The system interacts with S3 using the AWS SDK for Python (Boto3):

# src/config/s3/main.py import boto3 import asyncio from asgiref.sync import sync\_to\_async @sync\_to\_async def s3\_read\_bucket(bucket\_name: str): s3\_client = boto3.client('s3') objects = s3\_client.list\_objects\_v2(Bucket=bucket\_name, Prefix='data/') files = [] if 'Contents' in objects: for obj in objects['Contents']: files.append(obj['Key']) return files @sync\_to\_async def s3\_upload\_file(file\_path: str, bucket: str, object\_name: str): s3\_client = boto3.client('s3') try: s3\_client.upload\_file(file\_path, bucket, object\_name) return True except Exception as e: print(f"Error uploading to S3: {e}") return False

### Bucket Organization

Files in S3 are organized by file type and provider in a structured hierarchy:

bucket\_name/ ├── data/ │ ├── csv/ │ │ ├── affirm/ │ │ │ ├── file1.csv │ │ │ └── file2.csv │ │ └── bloomberg/ │ │ ├── file3.csv │ │ └── file4.csv │ ├── pdf/ │ │ └── ... │ ├── xml/ │ │ └── ... │ └── other/ │ └── ...

This organization is implemented in the upload\_file\_raw method:

async def upload\_file\_raw(self, file\_name: str, bucket\_name: str): sub\_folder = "other" if ".csv" in file\_name: sub\_folder = "csv" elif ".pdf" in file\_name: sub\_folder = "pdf" elif ".xml" in file\_name: sub\_folder = "xml" sub\_folder = sub\_folder + '/' + get\_folder\_name(file\_name, provider=self.provider\_name) await s3\_upload\_file('output/'+file\_name, bucket\_name, 'data/'+sub\_folder+'/'+file\_name)

The get\_folder\_name function determines the appropriate provider subfolder based on the file name and provider.

### File Tracking

The system tracks which files have already been processed by querying S3:

saved\_files = await provider.get\_save\_files(bucket\_name) saved\_files\_arr = [] for file in saved\_files: file\_array = file.split('/') saved\_files\_arr.append(file\_array[len(file\_array)-1])

This prevents duplicate processing of files that have already been uploaded to S3.

## EC2 Deployment

### Deployment Architecture

The Octo system is designed to be deployed on an AWS EC2 instance, as outlined in the SETUP.MD file:

┌────────────────────────────────────────────────────┐ │ EC2 Instance │ │ │ │ ┌────────────────┐ ┌────────────────┐ │ │ │ │ │ │ │ │ │ Docker │ │ SSH/SFTP │ │ │ │ Container │◄────────┤ Access │ │ │ │ (Octo System) │ │ │ │ │ │ │ │ │ │ │ └────────────────┘ └────────────────┘ │ │ │ │ │ │ │ │ ▼ │ │ ┌────────────────┐ │ │ │ │ │ │ │ AWS SDK │ │ │ │ (S3 Access) │ │ │ │ │ │ │ └────────────────┘ │ │ │ └────────────────────────────────────────────────────┘

### Instance Setup

The SETUP.MD file provides detailed instructions for setting up an EC2 instance:

1. **Instance Selection**:
   * Ubuntu 24.04 AMI
   * Appropriate instance type based on workload (t3.medium, t3a.xlarge, etc.)
   * Security groups configured for necessary access
2. **Docker Installation**:sudo apt-get install -y ca-certificates curl gnupg sudo mkdir -p /etc/apt/keyrings curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg # Additional docker installation steps...
3. **Code Deployment**:git clone https://github.com/<your-username>/<your-repo>.git cd <your-repo> git config credential.helper store git pull

### AWS Credentials Configuration

AWS credentials are configured in the EC2 instance to enable access to S3:

~/.aws/credentials

Or as environment variables:

AWS\_ACCESS\_KEY\_ID AWS\_SECRET\_ACCESS\_KEY

These credentials should be created with the principle of least privilege, with access limited to the necessary S3 buckets.

## IAM Roles and Permissions

### Required Permissions

The AWS credentials used by the system require the following permissions:

1. **S3 Permissions**:
   * s3:ListBucket: To list objects in the bucket
   * s3:GetObject: To download objects from the bucket
   * s3:PutObject: To upload objects to the bucket
2. **EC2 Permissions** (if using EC2 instance profile):
   * Permissions to access the required S3 buckets

### IAM Policy Example

A sample IAM policy for the Octo system:

{ "Version": "2012-10-17", "Statement": [ { "Effect": "Allow", "Action": [ "s3:ListBucket" ], "Resource": [ "arn:aws:s3:::${bucket-name}" ] }, { "Effect": "Allow", "Action": [ "s3:GetObject", "s3:PutObject" ], "Resource": [ "arn:aws:s3:::${bucket-name}/data/\*" ] } ] }

## S3 Bucket Configuration

### Bucket Creation

S3 buckets should be created with appropriate settings:

1. **Region**: Select a region close to the EC2 instance for optimal performance
2. **Access Control**: Enable appropriate access controls
3. **Versioning**: Consider enabling versioning for data integrity
4. **Encryption**: Enable server-side encryption
5. **Lifecycle Rules**: Configure lifecycle rules for cost optimization

### Security Best Practices

Follow these best practices for S3 security:

1. **Block Public Access**: Ensure all public access is blocked
2. **Encryption**: Use server-side encryption for all objects
3. **Access Logging**: Enable access logging to track bucket access
4. **Bucket Policies**: Implement restrictive bucket policies
5. **IAM Policies**: Use principle of least privilege for IAM roles

## Troubleshooting AWS Integration

### Common S3 Issues

1. **Authentication Errors**:
   * Check that AWS credentials are correctly configured
   * Verify that the IAM role has appropriate permissions
2. **Upload Failures**:
   * Ensure the bucket exists and is accessible
   * Check that the file exists locally before upload
   * Verify that the IAM role has s3:PutObject permission
3. **Bucket Listing Errors**:
   * Verify that the IAM role has s3:ListBucket permission
   * Check that the bucket name is correct

### Logging and Debugging

The system includes error handling for AWS operations:

try: s3\_client.upload\_file(file\_path, bucket, object\_name) return True except Exception as e: print(f"Error uploading to S3: {e}") return False

Enhanced logging can be implemented for better troubleshooting:

import logging logger = logging.getLogger(\_\_name\_\_) try: logger.info(f"Uploading {file\_path} to {bucket}/{object\_name}") s3\_client.upload\_file(file\_path, bucket, object\_name) logger.info(f"Successfully uploaded {file\_path}") return True except Exception as e: logger.error(f"Error uploading to S3: {e}", exc\_info=True) return False

## Future AWS Integration Enhancements

Consider these enhancements to improve the AWS integration:

1. **CloudWatch Integration**:
   * Send logs to CloudWatch for centralized logging
   * Create CloudWatch alarms for error monitoring
2. **SQS Integration**:
   * Use SQS for job queuing and distribution
   * Implement a more robust job system with retries
3. **Lambda Integration**:
   * Trigger processing based on S3 events
   * Implement serverless components for certain tasks
4. **RDS Integration**:
   * Store job history and metadata in RDS
   * Implement more robust job tracking

## Conclusion

The AWS integration in the Octo system primarily focuses on S3 for file storage and EC2 for deployment. The system uses the AWS SDK for Python (Boto3) to interact with S3, implementing a structured approach to file organization and tracking.

Developers working with the system should understand the AWS integration to effectively maintain and extend the system's capabilities while following AWS best practices for security and performance.

# Error Handling

## Introduction

Robust error handling is essential for ensuring the reliability and stability of the Octo system. This document outlines the error handling strategies implemented throughout the system, from API request validation to file processing errors.

## Error Handling Strategies

The Octo system implements multiple levels of error handling to ensure:

1. **Graceful Failure**: The system fails gracefully when errors occur, preventing cascading failures
2. **Detailed Logging**: Errors are logged with sufficient detail for troubleshooting
3. **Resource Cleanup**: Resources are properly cleaned up in error scenarios
4. **Progress Tracking**: Job progress is accurately reported, even when errors occur

## API-Level Error Handling

### Authentication Errors

The API uses FastAPI's dependency injection to handle authentication errors:

def api\_key\_auth(api\_key: str = Depends(oauth2\_scheme)): if api\_key not in api\_keys: raise HTTPException( status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Forbidden" )

When authentication fails, the API returns a standardized 401 Unauthorized response.

### Request Validation

FastAPI automatically validates request bodies using Pydantic models:

class RootBody(BaseModel): provider: str = 'affirm'

Invalid requests result in detailed validation error responses:

{ "detail": [ { "loc": ["body", "provider"], "msg": "field required", "type": "value\_error.missing" } ] }

### Duplicate Job Prevention

The API prevents duplicate jobs for the same provider:

existing\_jobs = [job for job in jobs.values() if job.provider == body.provider] if existing\_jobs: return {"message": "A job with the same provider is already in progress"}

This prevents race conditions and resource conflicts.

## Background Task Error Handling

The background task system includes error handling to prevent unhandled exceptions from crashing the server:

async def start\_new\_task(uid: UUID, param: int, provider: str) -> None: try: queue = asyncio.Queue() task = asyncio.create\_task(start\_logic(queue, provider)) while progress := await queue.get(): # monitor task progress jobs[uid].progress = progress jobs[uid].status = "complete" await asyncio.sleep(1) jobs.pop(uid) except Exception as e: print(f"Background task error: {e}") jobs[uid].status = "error" jobs[uid].result = str(e)

This ensures that:

1. Exceptions in background tasks are caught and logged
2. The job status is updated to reflect the error
3. The error message is captured for later reference

## Main Logic Error Handling

The main logic function includes comprehensive error handling:

async def start\_logic(queue: asyncio.Queue, provider\_str: str): try: # Main logic implementation # ... except Exception as e: print(f"an error occurred: {e}") await queue.put(None) return

This top-level try/except block ensures that any unhandled exceptions in the main logic are caught, logged, and reported back to the background task via the queue.

## File Processing Error Handling

The file processing loop includes error handling for individual files:

for file\_path in pending\_files: try: if file\_path.endswith('.sig'): continue # File processing steps # ... except Exception as e: count += 1 delete\_file(file\_path) percentage = (count/total\_pending) \* 100 await queue.put(float("{:.2f}".format(percentage))) files\_with\_error.append(file\_path) print(f"an error occurred while processing file {file\_path}: {e}") print(f"{bcolors.UNDERLINE}--------------------------------------------{bcolors.ENDC}") continue

This approach:

1. Catches and logs errors for individual files
2. Continues processing other files despite errors
3. Maintains accurate progress reporting
4. Tracks files with errors for later reporting

## Provider-Specific Error Handling

Provider implementations include error handling for their specific operations:

async def sftp\_download(self, file\_path: str): try: sftp\_client = paramiko.SFTPClient.from\_transport(self.ssh\_client.get\_transport()) print(f"Downloading {file\_path}") sftp\_client.get(f"inbox/{file\_path}", file\_path) sftp\_client.close() except Exception as e: print(f"Error downloading file {file\_path}: {e}") raise # Re-raise to be caught by the main processing loop

This allows for:

1. Provider-specific error logging
2. Proper resource cleanup
3. Error propagation to the main processing loop

## Resource Cleanup in Error Scenarios

The system ensures proper resource cleanup even in error scenarios:

try: # Operation that could fail finally: # Cleanup code that always runs delete\_file(file\_path) sftp\_client.close()

Key resources that are cleaned up include:

1. Temporary files
2. Network connections (SSH/SFTP)
3. File handles

## Error Reporting and Visualization

Errors are reported in several ways:

1. **Console Logging**:print(f"{bcolors.FAIL}Files with errors: {files\_with\_error}{bcolors.ENDC}")
2. **Job Status Updates**:jobs[uid].status = "error" jobs[uid].result = str(e)
3. **API Responses**:{ "uid": "123e4567-e89b-12d3-a456-426614174000", "status": "error", "progress": 75, "result": "Error downloading file file.txt: Connection refused", "provider": "affirm" }

## Error Types and Handling Strategies

### Network Errors

Network errors (connection failures, timeouts) are handled by:

1. Retrying operations (not currently implemented, but a recommended enhancement)
2. Logging detailed error information
3. Continuing with other files when possible

### File Format Errors

Errors in file formats (encryption, compression) are handled by:

1. Validating file extensions before processing
2. Providing detailed error messages for troubleshooting
3. Skipping problematic files and continuing with others

### Authentication Errors

Authentication errors (invalid credentials, expired keys) are handled by:

1. Immediate failure with clear error messages
2. Secure logging that doesn't expose sensitive information

### AWS S3 Errors

S3 interaction errors are handled by:

1. Exception handling around all S3 operations
2. Detailed error logging
3. Preventing partial uploads through proper error handling

## Logging Best Practices

The system implements several logging best practices:

1. **Error Context**: Logging includes the context of the error:print(f"an error occurred while processing file {file\_path}: {e}")
2. **Visual Differentiation**: Using color coding for different types of messages:print(f"{bcolors.FAIL}Files with errors: {files\_with\_error}{bcolors.ENDC}") print(f"{bcolors.OKGREEN}No files with errors{bcolors.ENDC}")
3. **Performance Metrics**: Logging includes timing information:print(f"{bcolors.WARNING}processed finished in {"{:10.2f}".format(elapsed\_time)} seconds {bcolors.ENDC}")

## Recommended Error Handling Enhancements

The current error handling system could be enhanced in several ways:

1. **Structured Logging**:
   * Implement a proper logging framework (e.g., Python's logging module)
   * Use structured logging formats (JSON) for better analysis
2. **Retry Mechanism**:
   * Implement automatic retries for transient errors
   * Use exponential backoff for network operations
3. **Error Aggregation**:
   * Implement a system to aggregate similar errors
   * Provide summary reports of error patterns
4. **Detailed Job Error States**:
   * Expand job status beyond "in\_progress", "complete", and "error"
   * Track specific error states (e.g., "authentication\_error", "network\_error")
5. **Error Notifications**:
   * Implement alerts for critical errors
   * Add email or webhook notifications for operational issues

## Conclusion

The Octo system implements a robust error handling strategy that ensures reliability and stability. By catching and handling errors at multiple levels, properly cleaning up resources, and providing detailed error information, the system maintains operational integrity even when issues occur.

Developers working on the system should follow the established error handling patterns, ensuring that new code maintains the same level of robustness and detailed error reporting.